CHAMELEON

by SEEQUA

USER'S MANUAL

SEEC
ORP

CHAMELEON
OWNERS MANUAL

DEMONSTRATION COPY
NOT FOR RESALE

SEEQUA
COMPUTER
CORPORATION

8305 Telegraph Road
Odenton, Maryland 21113
(301) 672-3600

COPYRIGHT NOTICE

All rights to this manual are reserved. No part of this document may be reproduced, transmitted, transcribed, stored in a retrieval system, or translated into any mechanical, magnetic, optical, chemical, manual, or in any other manner, without prior permission from SEEQUA Computer Corporation, 8305 Telegraph Road, Odenton, Maryland 21113.

© Copyright 1983 by SEEQUA Computer Corporation

DISCLAIMER

SEEQUA Computer Corporation reserves the right to make changes or improvements to the equipment, software, and documentation described herein at any time and without notice.

SEEQUA Computer Corporation disclaims any liabilities with respect to the quality, performance or failure of any software used on a SEEQUA product for any purpose or reason, or resultant loss of data.

TRADEMARKS

The following companies and their software, hardware, or documentation have been referenced in the following document. We hereby acknowledge their trademarks, registered trademarks, copyrights and all legal claim to the material.

Perfect Software, Perfect Writer, Perfect Speller, Perfect Filer, and Perfect Calc are trademarks of Perfect Software, Inc.

IBM and PC DOS are trademarks of International Business Machine Corp.

MS-DOS, MBASIC and MBASIC-86 are trademarks of Microsoft Corporation.

CP/M, CP/M 80 and CP/M 86 are registered trademarks of Digital Research, Inc.

Condor 1 and Condor 3 are registered trademarks of Condor Computer Corp.

This document is dedicated to the people who manufacture the SEEQUA Chameleon. In a business where pride of workmanship and outstanding quality often come second to profit, their dedication and skill have made the Chameleon one of the fastest rising stars in the microcomputer field.

TABLE OF CONTENTS

INTRODUCTION .....

FCC COMPLIANCE .....

CHAPTER

1 UNPACKING YOUR CHAMELEON

2 WHAT IS A CHAMELEON

Overview 2-1
The Computer Board 2-2
Power Supply 2-5
Fan 2-5
Disk Storage 2-5
Keyboard 2-7
Central Keys 2-7
Moving the cursor 2-8
Numeric pad 2-8
Return key 2-9
Control key 2-9
Special use keys 2-9
Function keys 2-10
Video display 2-10
Software 2-11

3 PROPER CARE AND FEEDING

Floppy Disks 3-1
Inserting Diskettes 3-2
Disk Drives 3-3
Video Display Unit 3-4
Keyboard 3-4
Main Unit and Power Supply 3-4
Software 3-6
Peripherals and options 3-7

4 SETTING UP YOUR CHAMELEON

Initial Set Up 4-1
Rear Panel (diagram) 4-2
Transporting the Chameleon 4-4

5 GETTING STARTED WITH MS-DOS

Booting a Disk 5-1
Using MS-DOS 5-1
Setting the Date 5-2
Setting the Time 5-3
The Keyboard under DOS 5-5
Copying a Disk 5-6
Comparing Two Disks 5-8
MS-DOS Files 5-10
Assigning Filenames 5-13
Global Characters 5-14

6 GETTING STARTED WITH PERFECT WRITER

Let's Get Going 6-1
Making Backup Copies 6-1
Using the Word Processor 6-4
Saving the Text File 6-6
Printing the Letter 6-8
How to use the Lessons Diskette 6-8
Other Chameleon Customized Features 6-9
Function Keys 6-9
Special Keys 6-11

7 GETTING STARTED WITH PERFECT CALC

Let's Get Going 7-2
Making Backup Copies 7-2
How to use the Lessons Diskette 7-6
Function Keys 7-7

8 GETTING STARTED WITH BASIC

In and Out of DOS 8-3
PROGRAMMING – The Indirect Mode 8-3
AUTOmatic Line Numbering 8-5
DEBUGGING – Your Syntax is Showing 8-6
Saving and Loading Programs 8-7
ENDing it All 8-9
BRANCHING – Or Where Do We GO TO From Here 8-9
IF..THEN – The Fork in the Road 8-9
The INPUT STATEMENT – Online Interaction 8-10
Run Disk, Run 8-11

TECHINICAL .....

APPENDIX – A ASCII CODE CHART

APPENDIX – B ASSEMBLY LANGUAGE INSTRUCTION SET 8088 & Z80A

8088 .....B-2
Z80A B-5

APPENDIX – C I/O PORT LOCATIONS

APPENDIX – D MEMORY MAP

APPENDIX – E I/O PORT PIN OUTS

Parallel port E-2
Serial port.....E-3

APPENDIX – F SAMPLE CABLES

Centronics Compatible Printers.....F-3
Daisywriter F-4

APPENDIX – G BIOS ENTRY POINTS

Commands G-2
ROM Bios Entry Points G-2

APPENDIX – H CTERM SMART TERMINAL SIMULATOR

General Description H-2
Function Key Description H-2
Buffer Description.....H-4

APPENDIX – I FILCOM

General Description I-2

APPENDIX – J SERDRV.OBJ

APPENDIX – K CHAMELEON BASIC86

APPENDIX – L OPTION

INDEX

INTRODUCTION

The Chameleon is a powerful, self-contained microcomputer system capable of performing a wide variety of computing functions for personal, business, and technical applications.

This manual is designed to provide you with the information you need to operate the Chameleon regardless of your previous computer knowledge.

Chapter 1 Unpacking Your Chameleon gives a brief rundown of what you should see when you first open the carton; what software, manuals, pamphlets and warranty cards have been included; how to get your Chameleon out of the box; and how to open it.

Chapter 2 What is a Chameleon tells the first time computer user what is in the Chameleon and introduces basic vocabulary used to describe microcomputers.

Chapter 3 Proper Care and Feeding is also for the first time user. It provides information on maintaining each of the basic components in the Chameleon. A little time spent learning about your equipment before you use it can prevent hours of frustration later.

Chapter 4 Setting Up Your Chameleon provides set-up information for the Chameleon and some of its more popular attachments. This chapter is written for a first time user, but also provides some schematic information that anyone adding attachments to the Chameleon will need to utilize, particularly if you are using options unique to the machine.

Chapter 5 Getting Started With MS-DOS contains basic instructions for utilizing the operating system that came with your Chameleon. While demonstrating some of the functions vital to the first time user of the Chameleon, it also includes information on unique features of interest to all Chameleon users.

Chapter 6 Getting Started With Perfect Writer is a quick introduction to word processing and use of the Perfect Writer software which was supplied free with your Chameleon. This brief outline will get you started using Perfect Writer in a hurry.

Chapter 7 Getting Started With Perfect Calc is another quick introduction to the world of spreadsheeting, informed forecasting and financial as well as statistical analysis. This is a powerful tool to add to your arsenal of 'success software'.

Chapter 8 Getting Started With BASIC will have you programming your Chameleon to do virtually anything you tell it in a matter of minutes. BASIC is a popular programming language for beginners and 'hackers' alike. If you've always wanted to know how computers do the things they do, then this is the perfect place to find out.

The Technical Section, which is made up of Appendixes, is a collection of specifications of the Chameleon and information concerning utilities, ports, cables, pinouts, ASCII Code tables, assembly language mnemonics and memory mapping.

As part of your reference manual, you will find two sections provided by Microsoft on BASIC and DOS. These are included as a guide to help you with areas of operation dealing directly with the software provided by Microsoft.

FCC COMPLIANCE

Warning: This equipment generates, uses and can radiate radio frequency energy and if not installed and used in accordance with the instructions manual, may cause interference to radio communications. As temporarily permitted by regulation it has not been tested for compliance with the limits for Class A computing devices pursuant to Subpart J of Part 15 of FCC Rules, which are designed to provide reasonable protection against such interference. Operation of this equipment in a residential area is likely to cause interference in which case the user at his own expense will be required to take whatever measures may be required to correct the interference.

CORRECTION PROCEDURES

If this equipment does cause interference to radio or television reception, which can be determined by turning the equipment off and on, the user is encouraged to try to correct the interference by one or more of the following measures:

- move the computer away from the receiver
- readjust the receiving antenna
- plug the computer or the receiving unit into a different outlet so that computer and receiver are on different electrical circuits.

If these efforts do not correct the situation, the dealer or an experienced computer, radio or television technician should be consulted.

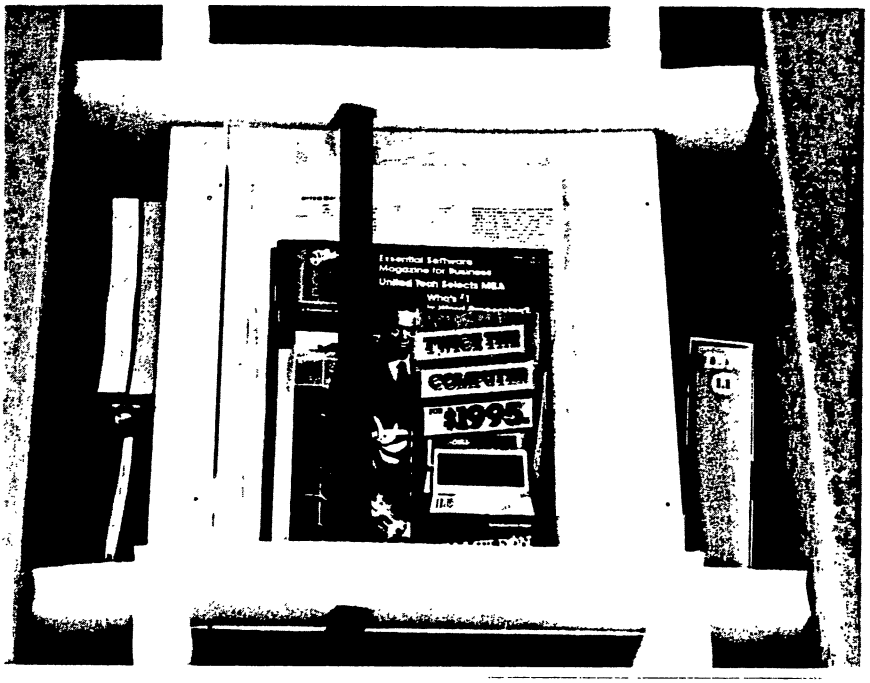
The Federal Communications Commision has prepared a pamphlet, "How to Identify and Resolve Radio-TV Interference Problems" (Stock No. 004-000-00345-4) which may also prove useful. It is available from the U.S. Government Printing Office, Washington, DC 20402.

WARNING

Shielded cables must be used with any peripheral devices attached to this computer to suppress radio energy interference and maintain FCC compliance. It is the responsibility of the user to abide by the characteristics under which this computer has been approved for operation.

CHAPTER 1
UNPACKING YOUR CHAMELEON

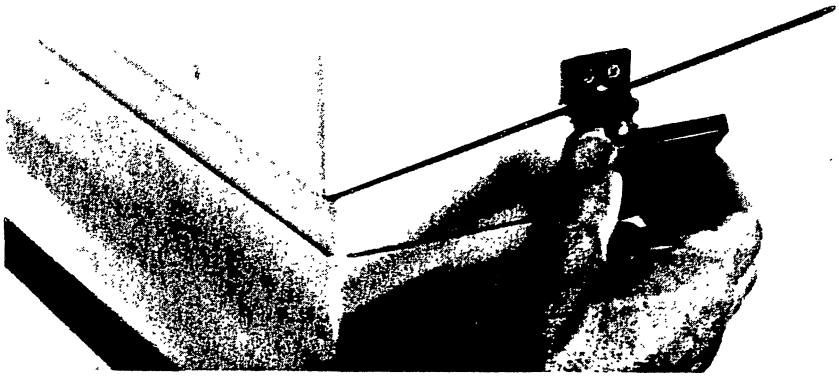
If you haven't already, open the top of the carton. You will see instruction manuals, including this one, and several diskettes in a plastic bag.



Place the plastic bag and other manuals aside temporarily. Use the handle to pull the Chameleon from the box. Remove the packing foam from either side of the Chameleon, and stand the Chameleon up with the handle on top.

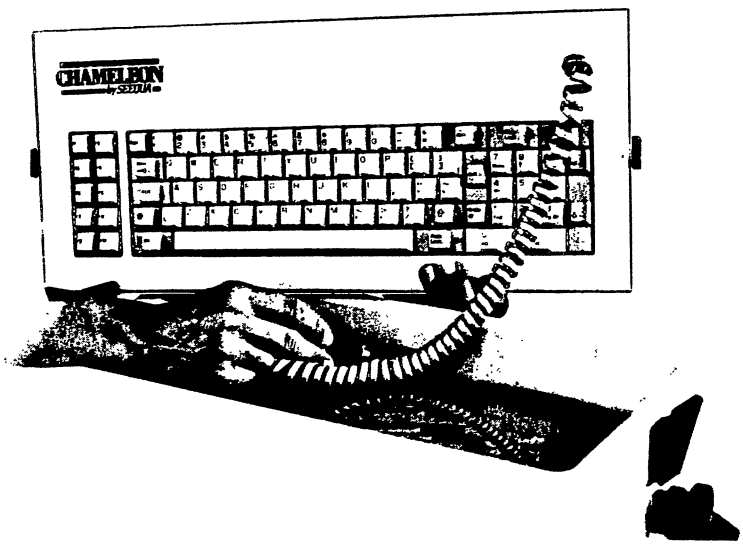
What you have in front of you is a Chameleon ready to travel. To open it:

- Loosen the two turn knobs on either end of the handle and drop the handle down to the side of the machine.
- Remove the keyboard from the main unit by releasing the two black latches on either side of the computer. This is accomplished by lifting up on the bottom of each latch.



Place the keyboard on a steady desk or table top.

- Detach the end of the keyboard cable from the velcro holder on the front of the Chameleon.



- Place the main unit next to the keyboard so that the back panel is easily accessible.

- Open the door on the back of the main unit. A wide, flat screwdriver works best for this purpose, but a dime can also be used.
- Inside the door is the power supply cord, the on/off switch, the brightness control knob, and a number of receptacles for various cables.

There it is. You have unpacked your Chameleon. Admire it and congratulate yourself on having purchased an amazing amount of computing power in one compact box.

You might want to take a few minutes now to check the other materials sent with your Chameleon. Inside the plastic bag is both a shipping list and warranty information. Make sure you received all that is on the shipping list, and fill out your warranty card according to the directions.

If you are familiar with computers and raring to go, turn now to *Chapter 4--Setting Up Your Chameleon.* Chapter 4 will tell you what all of those receptacles, knobs and switches are and how to use them.

If you are new to computers or feel you want to know more about the Chameleon and the software you have received with it, read *Chapter 2--What is a Chameleon and Chapter 3--Proper Care and Feeding.* The information in these two chapters should provide all the background information you will need to get started enjoying your computer today.

CHAPTER 2
WHAT IS A CHAMELEON?

OVERVIEW

The Chameleon is one of the most versatile and powerful computer systems contained in a portable package.

A computer system is a collection of electronic devices and logical instructions that receives data, translates it into electronic signals, processes it and displays the results. The electronic devices are commonly referred to as hardware and the logical instructions as software.

The Chameleon comes standard with an impressive array of both hardware and software. The standard hardware includes:

- The logic board containing two microprocessors, up to 48 K of read only memory (ROM), up to 256 K of random access memory (RAM), and various logic chips used to control input/output functions.
- Two disk drives for storage and retrieval of programs and information.
- A nine-inch, green video display unit, with an 80 character by 25 line display.
- An electronic keyboard.
- A power supply unit.
- A whisper fan for cooling.

Software that is provided as standard with the Chameleon includes:

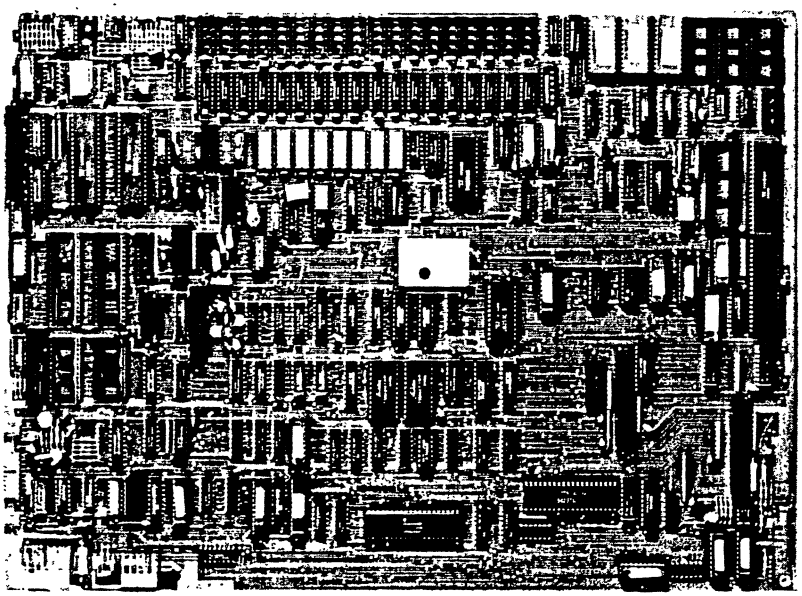
- System software stored in ROM.
- MS-DOS: a disk operating system.
- BASIC: a programming language.
- Perfect Writer: a word processing program.
- Perfect Calc: an electronic spreadsheet program.

The remainder of this chapter will explain what this hardware and software is and how it works together to perform various computing functions.

In addition to the above standard features, a wide variety of optional hardware is available, as well as an almost infinite variety of software.

THE COMPUTER BOARD

Inside the Chameleon is a printed circuit board. It measures 17" long by 12 1/2" wide and is covered with traces that conduct electricity between various integrated circuit chips.



Integrated circuit chips are made from rectangular pieces of silicon smaller than a fingernail. Large Scale Integration (LSI) technology has made it possible to place thousands of transistors on each of these chips. There are several different types of chips in the Chameleon, and they are used to perform many different functions.

The most complex chips on the board are the microprocessor chips. A ***microprocessor*** is designed to perform arithmetic and logic functions and control data transfer within the computer. A computer that uses a microprocessor to perform these functions is classified as a microcomputer. The Chameleon is called a dual processor microcomputer because it has two microprocessors: the ***Intel 8088*** and the ***Zilog Z80A.***

Before these microprocessors can process data, the data must be translated into an electronic form that the processor can understand.

The computer translates all data into on/off-type signals. These signals are referred to as ***bits***. In the Chameleon, as in most popular computers, it takes 8 bits to represent a single character such as the letter "A". Eight bits (or one character) of data is called a ***byte***.

The 8088 microprocessor chip is capable of processing 16 bits of data (or two bytes of data) at one time. This ability to process 16 bits of information makes data processing on the Chameleon extremely rapid.

The Z80A microprocessor chip processes 8 data bits (or one byte) at a time. The Chameleon, by having both the Z80A and 8088 microprocessors on the main logic board gives you two complete computer systems in one, allowing you to run a wide range of software programs.

Another type of chip found on the logic board is random access memory (***RAM***). These chips are used to store data within the computer. Data stored in RAM can be accessed by the microprocessors very rapidly (200 nanoseconds). This speed is relatively fast compared to the time it takes to retrieve data from other kinds of storage such as a disk drive. This is why most data currently being processed is kept in RAM. One important thing to remember is that RAM memory only retains data while electricity is supplied to the computer. If the power stops, either because you turn the computer off or because of a power failure, the data in RAM is lost and must be reentered. For this reason RAM is always used for temporary data storage.

The Chameleon comes standard with 128 K of RAM and can be expanded to 256 K on the internal logic board. The Chameleon Plus comes standard with 256 K of RAM. (***K*** represents 1024 bytes or characters of information.) This means that the RAM chips within the Chameleon can store up to 262,144 bytes at any given time. An additional 640 K bytes of RAM can also be added externally, with the Chameleon's expansion box option.

An important feature standard on the Chameleon, but found on few other microcomputers, is RAM ***parity checking***. Every time the Chameleon writes to its RAM memory, it also writes a parity bit. When memory is then read, the Chameleon checks the parity bit. If a discrepancy is detected, an error is issued to alert the operator to a possible memory failure. This feature is extremely important in assuring the accuracy of your data.

Another type of chip on the logic board is the read only memory (***ROM***). These chips are electronically preprogrammed before they are placed

on the Chameleon's board. They contain instructions which enable the Chameleon's microprocessors to accept data from the keyboard, display data on the screen, etc. Because these chips contain logical instructions, they are considered both hardware and software and are sometimes referred to as ***firmware***. The Chameleon contains 16 K of ROM standard and has room for an additional 32 K to bring the total to 48 K of ROM.

Other chips on the Chameleon's logic board perform interface functions. Some of these chips transform data that is in the format used by the microprocessors to the format needed to display the data. Others transform data from the incoming data format to the format needed by the appropriate microprocessor. These chips are all called *I/O* (input/output) interfaces.

The connectors, or ***ports***, of these interfaces can be seen on the rear panel. Those interfaces that come standard with the Chameleon have these ports. Those that are optional have blank covers in their respective locations.

I/O interfaces that come standard with the Chameleon are:

- monochrome and color composite video,
- the floppy disk controller,
- a serial keyboard interface,
- a video display controller,
- a parallel interface for adding a device such as a parallel printer,
- a serial interface for adding a device such as a serial printer or serial modem, (a modem allows the Chameleon to communicate with another Chameleon, or a data base on another computer).

Optional interfaces available for the Chameleon include:

- an analog to digital interface for digitizing, game paddles, joysticks, etc.,
- a bi-synchronous interface to add a device such as a bi-synchronous modem,
- an IEEE-488 interface that permits the addition of technical devices such as laboratory equipment,
- an external expansion box that provides eight slots for IBM PC compatible accessories,
- an RGB color display adapter.

POWER SUPPLY

There are several other components hidden inside the Chameleon's case aside from the logic board. One of them is the power supply unit. The power supply unit is an electronic mechanism that converts standard household electricity to the appropriate voltages and frequencies needed by the various Chameleon components. The power supply unit used in the Chameleon is a switching power supply designed to provide steady power, and to operate without generating heat that could interfere with the reliable operation of other computer components.

The Chameleon is set up to run on a standard U.S. power supply of 110 volts and 60 hertz. There is also an overseas power modification which will permit the unit to use electrical power transmitted at 220 volts and 50 hertz-the standard rates in most countries outside the U.S. If you will be using the Chameleon outside the U.S., ask your dealer to make the necessary modifications.

FAN

Another component inside the Chameleon's case is a whisper fan. This fan can be seen from the back of the computer. It is a simple electric fan designed to protect the system from any heat related problems.

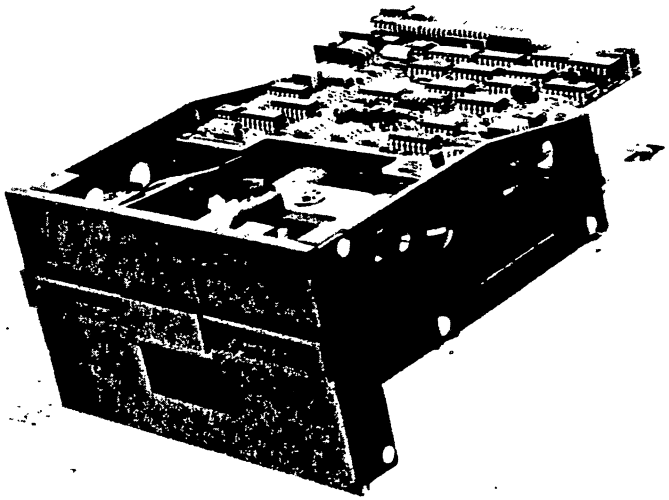
DISK STORAGE

The Chameleon comes standard with two disk drives. The drive on the left side (closest to the CRT screen) is the *A* drive. The one to the right is the *B* drive. The drives should always be referred to in terms of 'A' and 'B'. Each disk drive is capable of reading data from,or writing data to, a floppy diskette. These disks are used to provide mass storage.

Disk drives are mechanical devices that use a pressure pad and a spindle to position the disk, and a read/write head similar to a tape recorder to decode and encode information on the diskettes.

Unlike the RAM storage, once the floppy diskette has been encoded the data can be kept indefinitely whether current is being supplied to the disk or not. Furthermore, the disks can be removed from the drives and stored for a number of years before the data will deteriorate, thus offering convenient data storage offline, or outside the computer.

When the disk drives are in operation, a soft whirring noise can be heard. A red light is also provided on each drive to indicate that it is in use.



The disk drives provided on the Chameleon are ***single sided***. Each Chameleon disk drive is capable of storing 160 K of magnetically encoded data on a single side of a diskette.

With both disk drives this gives the Chameleon access to 320 K of mass storage at a given time.

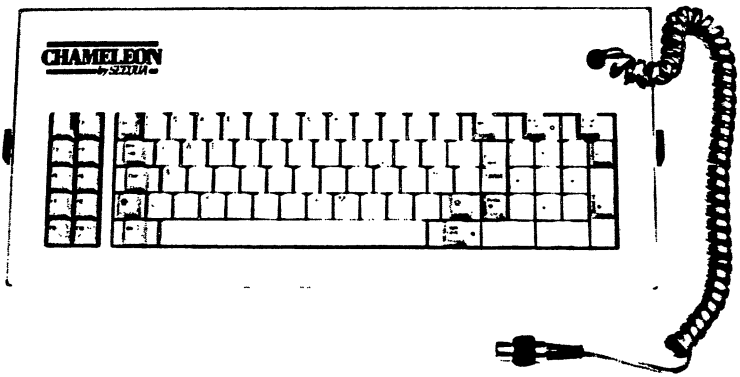
The Chameleon Plus comes standard with two doublesided, disk drives. These drives access both sides of a diskette and have access to 320 K of data each for a total storage capacity of 640 K.

The disk drives supplied with both the Chameleon and the Chameleon Plus are ***dual density***. Density refers to the amount of data that can be maintained in a given amount of disk space. The Chameleon disk drives are designed to read ***5 1/4-inch floppy diskettes***. A single density 5 1/4-inch diskette can hold from 80 K to 90 K of data on one side. The Chameleon dual density drives can store and retrieve 160 K on one side of the 5 1/4-inch diskettes.

Be careful to purchase the appropriate diskettes for your Chameleon. Ask for single-sided, dual density, soft-sector 5 1/4" diskettes for the Chameleon and double-sided, dual density, soft-sector 5 1/4" diskettes for the Chameleon Plus.

KEYBOARD

The Chameleon keyboard is used to input data and issue commands to the computer. There are 83 keys on the Chameleon keyboard. Each time any of the keys is struck, an electrical impulse is sent to the keyboard I/O interface. How that impulse is used and what each key represents is controlled by the software in use at the time.



A special feature provided on the Chameleon keyboard are LED's. ***LED*** stands for light emitting diode. These are the two red lights on the 'Num Lock' key and the 'Caps Lock' key. When these keys are struck, the appropriate LED glows to remind you that either the numeric or upper case mode is activated.

The following section gives a description of how all the keys are generally used. The manuals provided with individual software programs will explain uses of individual keys that are specific to a particular software program.

Central Keys

The central part of the Chameleon keyboard is similar in appearance and use to a standard typewriter keyboard:

- The long white bar at the bottom is a space bar.
- The white lettered keys are used to key both upper and lower case letters.
- The white numbered keys across the top are used to key the numbers shown.

- The following symbols can be keyed when in lower case mode:

\ , . / ; ' ` [ ] - =

- The symbols shown above the numbers and the above keys can be keyed using the shift key:

| ( ) ? : " ~ { }

! @ # $ % ^ & * ( ) _ +

The gray keys with open vertical arrows (⇳) on either side of the central keys on the second row up are the shift keys. They are used to capitalize a letter and to key the special characters shown.

- The gray 'Caps Lock' key on the bottom row is similar to a shift lock key on a typewriter and can be used to key a series of capital letters. Unlike a typewriter, the lock cannot be removed by using the shift key. To end the lock, re-key the 'Caps Lock' key itself.

MOVING THE CURSOR

Another seemingly familiar part of the keyboard are the 9 white keys arranged in a square at the right hand side of the keyboard. Generally, these keys are used to control the movement of the cursor.

- The 'Home' key puts the cursor at the top left hand corner of the screen.
- The 'Pg Up' key will scroll the screen up one full screen.
- The 'End' key is often used to scroll to the last data in a file.
- The four keys with arrows are each used to move the cursor one space in the direction the arrow is pointing. To move the cursor more than one space in any direction, simply hold the key down and the cursor will continue to move in the direction indicated until the key is released.

NUMERIC PAD

The nine keys used for cursor movement can also be used for a numeric pad to enter numbers in a manner similar to using a calculator. To use the keys for numerics, depress the 'Num Lock' key. Like the 'Caps Lock' key, the 'Num Lock' key is used both to activate and deactivate the

numeric pad. Other keys are also affected when the 'Num Lock' key is activated.

- The '−' found to the right of the number nine can be used to enter a minus sign.
- The '+' found to the right of the numbers six and three can be used to enter a plus sign.
- The '*' found to the left of the number one can be used to indicate multiplication in a formula.
- The '0' under the '*' and number one is used for zero.
- The period under the numbers two and three is used to indicate a decimal point.

RETURN KEY

One of the most important keys to become acquainted with is the ***Return*** or ***Enter*** key. It is the longer gray key with the left-facing arrow just to the left of the numbers seven and four on the numeric pad. This key is used to send information to the Chameleon when you have finished keying a command or program information.

CONTROL KEY

The control key is the long gray key to the left of the letter 'A' key, labeled ***Ctrl***. The 'Ctrl' key is always used simultaneously with another key. The 'Ctrl' key serves as a signal that the key being used with the 'Ctrl' key is doing more than just entering the value of that key. It is part of a functional command.

SPECIAL USE KEYS

- The 'Esc' key is the gray key found on the top line, just to the left of the number one. It is often used to interrupt a command.
- The key directly below the 'Esc' key is the tab key. This is generally used like the tab key on a typewriter.
- The 'Alt' key, below the left shift key is used similarly to the 'Ctrl' key, as it is used with another key to change the meaning of the other key being struck at the same time.
- The gray key to the left of the 'Num Lock' key is the backspace key. This will generally delete the last character keyed.

- The 'PrtSc' key located below the Return key is used to print the information displayed on the screen.
- The 'Scroll Lock' key is used by some software to permit you to enter data beyond the established margins. When used with the 'Alt' key, it will generally cause a break in a command.
- The insert key, 'Ins', is found below PrtSc and '1' on the numeric pad. This is used to insert characters within a line.
- The delete key, 'Del', is to the right of the Ins key. It is used to erase the character to the left of the cursor.

FUNCTION KEYS

The ten gray keys on the left side *(F1 - F10)* are function keys. These keys are provided to permit single stroke entry of frequently used commands. Both these function keys and the special editing keys have different uses depending on the program being used. For the uses of these keys in MS-DOS, Basic, Perfect Writer, Perfect Calc, etc., refer to the manual provided with these programs.

VIDEO DISPLAY

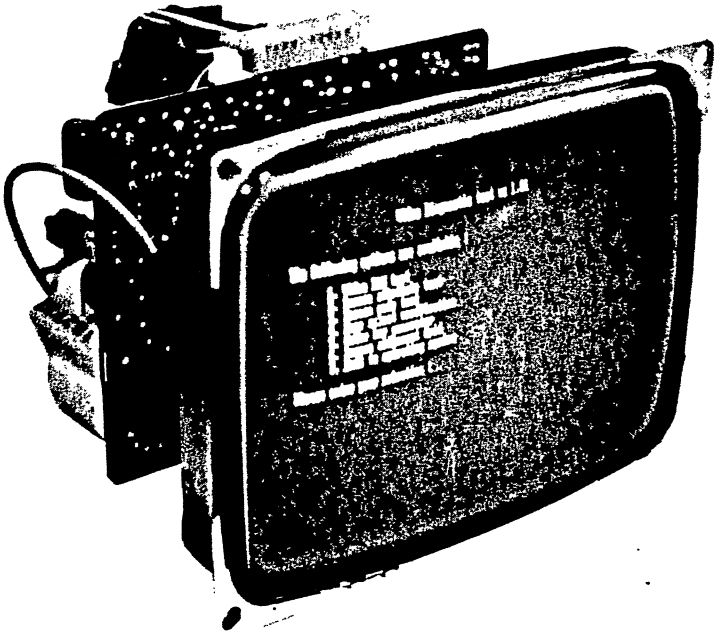
The video display unit provided standard with the Chameleon is a cathode ray tube providing a green phosphor display on a black background. A display unit is commonly called a *CRT* or a tube.

The Chameleon's standard CRT measures 9 inches on the diagonal. Hence it is a 9-inch screen. It displays 80 characters across and 25 lines down. The bottom line is a system status line and cannot be utilized for user display. Each character displayed on the screen is made up from a group of dots. The matrix for depicting each character is 8 dots across and 8 dots down.

When graphics packages are used, the screen can display up to 640 dots across by 200 dots down. This provides the high resolution necessary to portray clear graphic displays.

By attaching a color monitor, full color display capabilities are available. In text mode sixteen distinct colors can be displayed. They are black, blue, green, cyan, red, magenta, brown, light gray, dark gray, light blue, light green, light cyan, light red, light magenta, yellow, white.

In graphics mode any four of eight colors can be used at one time. Color graphics are depicted with 320 dots across and 200 dots down. Characters on the color screen utilize the 8 X 8 dot matrix.



SOFTWARE

Software is the name for the instructions that tell the computer what to do. Software comes in many varieties. Some software is permanently encoded in the Chameleon's ROM chips. Some software is called ***systems software*** and deals primarily with general system functions such as controlling input and output operations and making sure that various parts of the system communicate with each other properly. ***User software*** or ***applications software*** are generally programs designed to perform tasks for home or business use. Games are also included in the generic term 'user software'. Without software, the Chameleon would not function.

Software is comprised of commands and functions. Commands are instructions that the computer can translate and perform. Programs are sets of commands designed to perform a particular function or group of functions.

There are various languages used to express computer commands. Some of the more popular programming languages are: COBOL, BASIC, ASSEMBLER, PILOT, PASCAL and FORTRAN. Once you have learned to use any of the languages compatible with the Chameleon you can issue commands and create your own programs.

There are also many packaged programs available to perform particular functions. These programs are generally stored on a floppy diskette and need only to be read into RAM from the disk drive to be used.

One of the most important items of software is the ***operating system***. The operating system is a set of programs that performs the housekeeping functions needed to run a computer. It controls data transfer functions, file control functions, data display functions, etc. The operating system allows you to issue basic operating instructions so you can input data, output data, copy disks, etc. Several operating systems can be used with the Chameleon--the more popular ones are: MS-DOS, IBM PC DOS, CP/M 86, and CP/M 80.

A wide variety of software is available to allow you to use the Chameleon for various tasks from income tax analysis to playing games. To get the most from your software, ask your local dealer if specific programs will run on the Chameleon; and before using the software, aquaint yourself thoroughly with the manuals provided.

The Chameleon comes standard with several software programs:

- MS-DOS, an operating system that allows you to use the Chameleon in a manner that is compatible with an IBM PC using PC DOS.
- BASIC, a simple programming language that will permit you to write your own programs.
- Perfect Writer, a set of programs that allow you to use the Chameleon as an efficient word processor.
- Perfect Calc, a set of programs that allow you to use the Chameleon to analyze various spreadsheet situations.

Software is often available in both 8- and 16-bit versions. All of the software provided standard with the Chameleon is designed for the 16-bit, 8088 microprocessor. This provides you with maximum operating efficiency.

CHAPTER 3
PROPER CARE AND FEEDING

Your Chameleon is manufactured from quality components, and is designed, assembled and tested to insure that it will:

- give dependable service,
- travel with a minimum of effort,
- function well in a variety of work environments.

Nevertheless, it is important that you remember a few rules regarding the care and servicing of the Chameleon. The following chapter will briefly describe important components in your Chameleon and give some suggestions regarding their care.

FLOPPY DISKS

The Chameleon uses 5 1/4-inch disks or diskettes to permanently store programs and data items. These disks are fragile and damage to them can result in loss of all the data stored on the disk or in a partial data loss which often is the real culprit when programs seem to be malfunctioning. Much frustration will be avoided if your disks receive proper care.

Floppy disks are sensitive to electromagnetic fields. They should be kept away from magnets and magnetic coils such as those in telephones and televisions. They should be kept away from the x-ray machines found in airports and radio communication equipment.

Data on disks can be damaged by the oil and dirt on your hands. Take care not to touch the exposed area of the magnetic material. Always grasp the diskette on the area of the disk where the label is attached.

Both the diskette itself and the disk drive can be damaged by dust on the floppy disk. The floppy disk should be kept in its paper jacket when not in use and stored in boxes or special plastic holders that are designed to go in notebook type binders. Visually inspect each diskette before inserting it in the drive and remove any dust by blowing carefully on the diskette. Smoke particles can cause disk damage as well as the dust particles that can be seen. Smoking should be minimized in the rooms where the computer is going to be used and/or the floppy disks stored.

Liquids should never be permitted near a floppy diskette. A floppy diskette that has been exposed to any kind of liquid should be considered totally ruined and should be discarded.

Floppy disks can also be damaged by extreme temperatures. Normal operating temperatures range between 10 to 50 degrees Celsius or 50 to 122 degrees Farenheit. Diskettes that have been exposed to temperatures below 5 degrees Celsius or 40 degrees Farenheit should be discarded. The floppy disks should also be kept out of direct sunlight.

Diskettes that have been creased or punctured should not be used. It is particulary important that a floppy disk that is going to be shipped in the mail be carefully packaged and protected with strong cardboard so it will not be damaged in transit.

Floppy disks will warp. To prevent this, always store the diskettes in proper containers and in an upright position. Never stack diskettes on one another or allow them to lie on heat generating equipment.

Data can be damaged by pressure on the diskette. Always grasp the diskette on the labeled area and hold the diskette gently. When labeling a floppy disk always use a felt tip pen. Never use a ball point pen or a pencil.

INSERTING DISKETTES

It is important that the floppy disk be placed in the disk drive properly. Although the diskette is square and can physically be inserted in several ways, for a one sided disk there is only one proper way. Attempting to read an improperly inserted diskette can cause damage to the data and possibly the drive. To insert the diskette open the door to the drive. Hold the diskette with the thumb of your right hand on the floppy disk label. In order to insert the diskette you must hold it vertically so you can see the label when it is in your right hand and gently put it in the slot of the appropriate disk drive. You will hear a slight click and be able to feel when the diskette is fully inserted. Once the diskette is properly positioned, close the door to the disk drive. Doors on the disk drives should never be opened or closed when the red light is on or the whirring sound of the drive motor can be heard.

In spite of all your care, you will still probably experience problems with deteriorating or damaged floppy disks. Inspect diskettes before using them for obvious damage, but more importantly, make backup copies

of all important diskettes. Clearly label the backup diskettes with the date and an indication of the contents of the diskette. How to copy a diskette is covered in *Chapter 5* of this manual.

It is possible to delete files or copy over files unintentionally and consequently destroy valuable data. To help prevent these kind of errors, always label your diskettes and if possible keep a listing of the contents with the diskette. Most floppy disks have a notch on the right hand side of the diskette. This is a write-protect notch. When it is covered with one of the metallic tabs provided with the diskette, the drive will not write on the diskette. Software locks can also be used to protect particular files on a diskette from being written over or deleted. More instruction on these techniques are given in *Chapter 5* of this manual. Remember that a little organization can prevent hours of frustration.

DISK DRIVES

The Chameleon uses disk drives to transfer information from the floppy disks to memory and from memory to the floppy disks. With proper use and simple routine maintenance these drives should give years of trouble free service.

Open and close the disk drive door gently. When not in use the doors should be left closed. Never open the disk drive door when the ***in use*** light is on or you can hear the whirring of the motor. To stop the motor you must reset. This is done by turning off the power to the main unit or pressing the ***Ctrl, Alt, and Del*** keys on the keyboard simultaneously.

Insert and remove floppy disks gently. You will hear a soft click when the disk is in position. Never force a diskette into the drive. If it doesn't slip in easily, there is probably already a diskette inserted. Remove it and try again. If there is not another diskette in the drive and it continues to jam, take it to your local dealer. Never insert anything other than a 5 1/4-inch floppy diskette into the Chameleon disk drive.

Protect the drive from dust and smoke by using the Chameleon in a clean environment. If you are not going to use the Chameleon for quite a while, cover it to protect the drives from environmental dusts.

The drives are also sensitive to shocks and vibrations. They should not be dropped or used on an unsteady surface.

Several times a year, the read/write heads should be cleaned. This is done by using a special floppy disk that is soaked with cleaning solvent.

These should be available at your local dealer. If your Chameleon is used constantly you will want to clean the heads more frequently. This is the first step to take if read or write errors occur.

Occasionally, in spite of all precautions, the heads may need to be calibrated and aligned. This must be done by your local dealer.

VIDEO DISPLAY UNIT

The video display unit or CRT (cathode ray tube) is used to display information for the Chameleon operator. The CRT is rugged and should be trouble free.

The brightness control, located on the back of the Chameleon, is the only mechanical adjustment. The brightness is adjusted properly when it is comfortable for the operator.

The screen should be kept clean. Again this is for your comfort. When necessary the screen can be cleaned with a soft, lint-free cloth. If that is not sufficient, a damp cloth and mild detergent can be used, but be extremely careful not to use a *wet* cloth and run the risk of damaging more fragile system components.

KEYBOARD

The keyboard is used to send information to the Chameleon. It is durable and should be trouble free. It can be damaged by liquids, dust, dirt and physical impact, but it is not nearly as vulnerable to these agents as the floppy diskettes and disk drives.

Care must be taken not to drop the keyboard and to insure that it is securely fastened to the main unit for transport. Keyboard maintenence is simply that of periodic cleaning. The best method is to use a can of pressurized air which can be purchased at your local dealer. If used carefully, a vacuum cleaner can also be used to dust the keyboard.

Occasionally, static electricity will interfere with the electrical impulses sent by the keyboard. If this seems to be a problem in your environment, anti-static mats can be purchased to place under the operator.

MAIN UNIT AND POWER SUPPLY

Inside the Chameleon is a complex array of memory chips, I/O interfaces, microprocessors, a fan and a power supply unit. These components execute programs, store data, communicate with the external system

components, provide a clean source of power for the unit, keep the temperature cool for prime operating conditions--in short do most of the work.

The Chameleon is made of high quality components and is designed and assembled for durability and trouble free use. Nevertheless, the main unit should be treated with care and respect.

Handle the Chameleon gently. Don't drop it and don't shake it. If you must swear at it, do so, but physical abuse will not be rewarded. Be especially careful about this when transporting the Chameleon. The Chameleon is a portable unit, but it is not a piece of luggage, and should not be treated as one.

The Chameleon should be placed on a steady stand, table or desk. Vibrations occuring while the Chameleon is working could affect its performance.

Protect the Chameleon from dust, dirt, smoke and liquids, by keeping smoke and beverages away from the Chameleon's work area. The Chameleon should be kept clean, particularly the vent area on the back of the main unit. Dusting can be done with pressurized cans of air packaged for these purposes or by a gentle dusting with a vacuum cleaner. Don't put a full power vacuum cleaner directly on the Chameleon; hold the attachment a little off the surface.

Keep the Chameleon cool. The internal components work best at temperatures around 60 to 70 degrees Farenheit. The Chameleon has its own fan to reduce the heat generated by the computer components, but this will not compensate for a hot computer room. If intermittent errors occur, and there is a reason to suspect high temperatures as a cause, turn off the Chameleon, and let it cool.

One of the most important elements for proper functioning of the Chameleon is a steady, consistent power supply. The Chameleon is equipped with a power supply unit that will convert 110 volts current at 60 hertz or as an option 220 volts at 50 hertz to the voltages required by the Chameleon. This unit will compensate for minor fluctuations in the current provided by the utility company. Major fluctuations, however, will cause problems with the Chameleon's operation, and in some cases could cause permanent damage.

There are a number of things you can do to minimize the possibility of this type of damage. The simplest and most important is to set aside a

separate electrical circuit for the main unit of the Chameleon. Label this circuit on the electrical panel in such a way that it will not be turned off when other electrical work is done. If this is not possible, make certain that no major appliances are sharing the power. If you have a printer for the Chameleon, make sure they are on separate circuits. If you are unsure about what is on a particular circuit, you can switch the circuit breaker off and see what loses power. Do this experimentation when the Chameleon is off.

If you have taken these precautions and still have problems that could be related to the power supply, try testing the voltage at different times of day and insuring that it is fairly regular. If it is not regular or is not providing the required voltage, there are several intermediate devices that can be purchased to clean the supply power before it reaches the Chameleon. The Chameleon battery pack can perform this function.

Another important step you can take is to make sure that the connections to the Chameleon are grounded. This will prevent damage to both you and the Chameleon. If the Chameleon is to be used in a business environment, two grounds should be provided: one for reference (normal ground in plug) and another one for safety (grounding wire connected directly to the Chameleon's chassis).

Power interference can also come from static electricity. Static electricity can be reduced by eliminating static producing items such as wool and nylon rugs. It can also be reduced by increasing the relative humidity in the room. If problems persist anti-static mats can be purchased and placed under the Chameleon.

SOFTWARE

Software is the general term for the instructions that tell the computer to perform. Proper care of the software is vital to both the user and the system to maintain a smoothly functioning system.

As a novice computer user, you will be largely dependent on software provided by others. The floppy disks provided with the Chameleon are examples of user software. These disks contain programs that allow you to enter and manipulate data for word processing, spread sheet analysis and data and system maintenance. The use of these programs will be explained in an accompanying manual. However, as a novice user it is important to be aware of the software and treat it with respect.

The first step in software maintenance is to make copies of all important software. See *Chapter 4* for instructions on how to make backup copies. Make sure all copies are labeled with both a description and date. Equally important is taking the time to review the manuals provided so you know what to expect from the software you intend to use.

Various programs will change the functional characteristics of the Chameleon. The most common change is in the use of the various keys on the keyboard. Other changes could be the prompt on the CRT or the print characteristics of the printer. Make notes on these kind of changes until you are familiar with the programs, and keep them handy.

Particular care must be taken with functions that could cause an accidental loss of data. Most problems with computer systems stem from errors made by operators who are unfamiliar with the operating characteristics of the software being used.

If there seems to be a discrepancy between the program and the instruction manual, write down what you expected the software to do, what it actually did and any other facts you can recall about general system performance that might help track the problem. Make sure you include date and time of use and any version number that might be displayed for the software you are using. If you have a backup of the software, try repeating the series of events using the backup. If this does not resolve the problem, report these discrepancies to your local dealer. If the dealer cannot explain them to you, or resolve the discrepancy, he/she will use your notes to report the problem to the software supplier and try to get an answer for you.

Once you start creating your own programs or changing programs that have been supplied to you, both your system's flexibility and your responsibility increase. The important thing to remember once you start programming is to keep organized records to track both system performance and changes made to programs. Dates should be noted when any changes are made, and notes kept on the type of change, and your reason for making the change. Backups of a program should always be made before changes are made and kept on hand until proper performance on the new software is completely assured.

PERIPHERALS AND OPTIONS

The Chameleon expansion box contains components similar to the ones in the main unit, and it should receive the same kind of care. Protect it

from dust, dirt and liquids. If it is moved, take care not to drop or jolt it. While the Chameleon is in operation place the box where it will not be subject to vibrations. If servicing is necessary, consult your local dealer.

The Chameleon can be operated with a variety of devices that are purchased from manufacturers other than Seequa. Make sure you receive an operations manual with any device you purchase, and acquaint yourself with it thoroughly before attaching it to the Chameleon. In general, purchased peripherals will benefit from the same type of care techniques discussed previously: clean power, clean environment, cool temperatures and no liquids on the equipment.

A few special precautions should be taken when using a printer or a color TV as peripherals. Both of these devices consume a fair amount of power, and should not be plugged into the same electrical circuit as the Chameleon. The printer will be the device with the most moving parts and therefore the most susceptible to environmental hazards. Furthermore, a printer can be seriously damaged by a paper jam that is not corrected immediately, so it should never be left printing unattended.

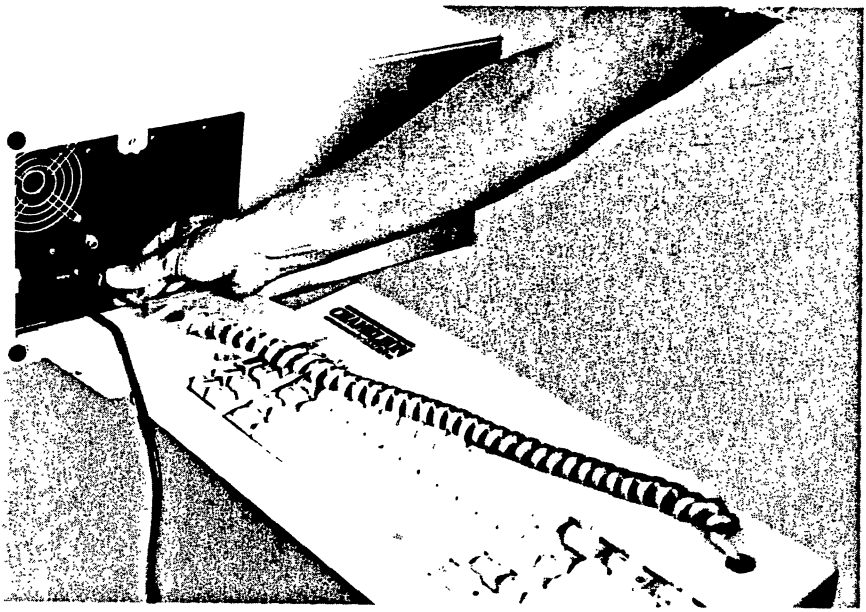
A color TV can cause damage to data stored on floppy disks. Care should be taken to keep the TV as far away from the disk drives and floppy disks as possible.

CHAPTER 4
SETTING UP YOUR CHAMELEON

INITIAL SET UP

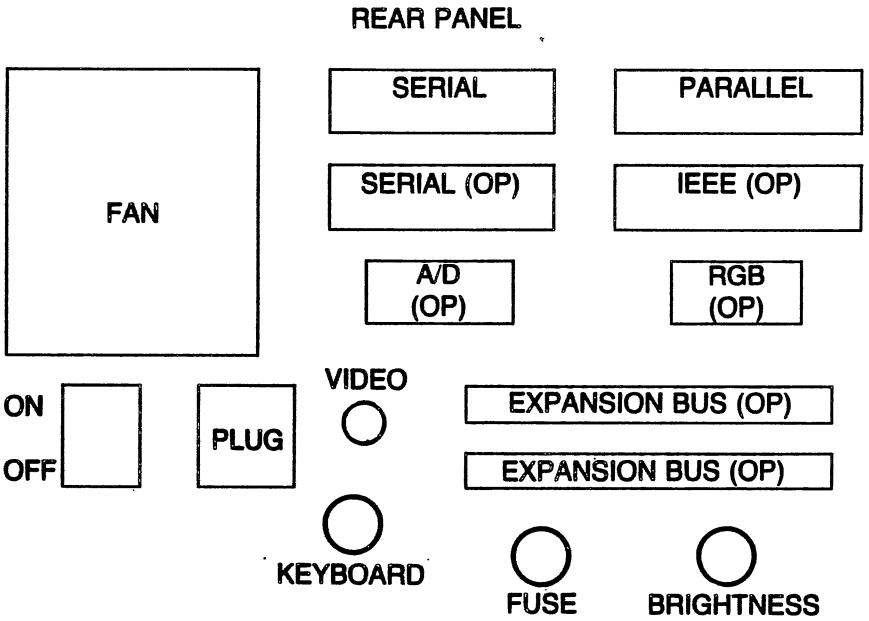
A few simple steps must be taken before operating the Chameleon:

- Remove the keyboard from the main unit by releasing the two black latches on either side of the system.
- Carefully place the keyboard on a steady desk or table top in a comfortable operating position.
- Place the main unit next to the keyboard so that the back panel is easily accessible.



- Open the door on the back of the main unit by turning the screw at the top of the door. A screwdriver works best for this.
- Remove the power supply cord from the cavity and take a few minutes to familiarize yourself with this panel.

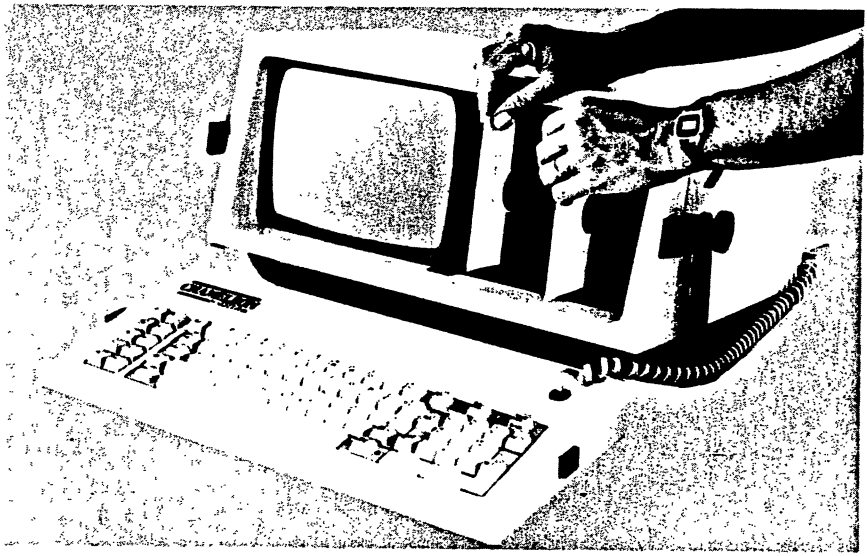
The following diagram identifies each item that can be on the panel. Usually your Chameleon will have several slots that are not filled. These slots will be used to insert connectors for dealer installed options (OP).



Now that you are familiar with the back panel on your Chameleon, continue with the following set up procedures:

- There is a cable attached to the keyboard. If it is still attached to the velcro holder on the front of the Chameleon, detach it.
- Note the prongs in the plug at the end of the cable, these prongs fit into corresponding holes in the top of the keyboard socket on the back panel. Plug the free end of the cable into the keyboard socket so the prongs are aligned. NOTE: The power should always be turned off when you attach or detach the keyboard cable.
- Turn the brightness control knob clockwise to ensure that the video display will be easily seen.
- Make sure the on/off switch is in the *off* position by pressing downward on the switch. (That's the bottom of the switch pressed inward.)
- Extend the power cord and plug the female end into the appropriate receptacle on the Chameleon's back panel. (See preceding diagram.)

- Place the main unit on your desk or table top behind the keyboard with the screen on your left and the disk drives on your right.
- Place the carrying handle under the main unit; this will serve as a footing for the Chameleon.
- Gently open the door to the disk drives, remove the piece of cardboard from each drive and close the disk drive doors.



Note that the cardboard is used to protect the disk drive heads when the Chameleon is being shipped or transported. You should save these and use them when you transport the Chameleon.

You are almost ready to supply power to the Chameleon. First test the electrical outlet by plugging in an ordinary lamp, radio or small electrical appliance to make sure the outlet is functioning properly. Next, unplug the test appliance and plug in the Chameleon's power supply cable.

After you have plugged the cord into the electrical outlet, move the on/off switch to the ***on*** position. You should see a blank screen with a small green line in the upper left hand corner. Within a few seconds you will hear a short beep. This tells you that the Chameleon is set up correctly and is ready to accept a diskette.

If you have purchased additional equipment to use with the Chameleon, it will be attached to the Chameleon's back panel. To attach equipment,

you must have the proper cable to connect the equipment to the appropriate receptacle on the panel. (See preceding diagram.)

One serial and one parallel connection come standard with the Chameleon. If you purchase equipment that requires optional connectors, have your local dealer install them for you and provide you with the appropriate cable to connect your equipment.

Consult the manual provided with the equipment for specific set up and operating instructions for each device you use.

TRANSPORTING THE CHAMELEON

The Chameleon is designed to be a self-contained portable unit. To ready the Chameleon for a safe trip takes just a few seconds.

- First, disconnect the cables that connect the keyboard, and other peripheral devices to the back panel of the Chameleon.
- Place the power supply cord in the back panel pocket, fasten the back panel door using a screw driver or a small coin, and turn the lock clockwise.
- Insert the protective cardboard packing pieces into the disk drives and close the disk drive doors.
- Stand the Chameleon up with the back panel on the bottom and the handle on top.
- Attach the keyboard cable to the small piece of velcro on the front (top) of the Chameleon. This will keep the cable end from damaging the screen on the video display unit.
- Place the keyboard on top of the Chameleon in an upside down position so that the Chameleon and keyboard form a rectangular box.
- Latch the keyboard to the Chameleon, and it is ready to go.
- If you have purchased the protective carrying case for the Chameleon, simply use the handle to place the Chameleon in its case. This case will help protect the Chameleon from getting scratched, but is not necessary for safe transport of the inner parts.

The Chameleon is now ready to go. Because of its single board, chip architecture, it is reasonably durable. However it should be treated with some care. It is not a piece of luggage and should not be shipped unless it has been repacked in its original shipping case. It must also not be subjected to surveillance x-rays or taken in the vicinity of other strong electro-magnetic fields.

CHAPTER 5
GETTING STARTED WITH MSDOS

BOOTING A DISK

To operate a computer both hardware and software must be used. In order to place a particular software program in the Chameleon, you must ***boot*** or load the software from a disk.

Booting a disk can be accomplished by two methods--the Cold Start or Initial Program Load (IPL, for short), and the Warm Start or reset procedure.

To boot software from a Cold Start, make sure the Chameleon is turned off; place an appropriate disk in drive A, the drive to your left; and then press the power switch on the rear panel to the ***on*** position.

If the Chameleon is already turned on, you can do a warm start to load or re-load software when needed. Place an appropriate disk in Drive A, then press the ***Ctrl*** key, the ***Alt*** key and the ***Del*** key ***simultaneously,*** hold them for a moment, and then let go of all three keys.

After performing either of the above procedures, the computer will go through its internal start-up procedures. The red light on drive A will light and drive A will begin making whirring noises. At this point, the system will beep and one of two events should occur. Either the disk will boot automatically or a message similar to the following will be displayed.

Chameleon Monitor Version x.x
Copyright Seequa Corp. 1983

In the second case you must type **b** and press the Enter key to boot, and this will cause drive A to read and load the disk.

USING MS-DOS

Some software, such as the IBM Adventure Game, can be booted and used without the benefit of an operating system. However, most software is designed to run under the direction of an operating system which must be loaded into the computer's memory before it can be used.

The preferred operating system for the Chameleon is MS-DOS (pronounced 'emm ess doss'). Almost all work sessions will start by loading or booting MS-DOS.

Practice booting your MS-DOS diskette using the two methods just described. Notice that the red light on the left drive is lit while it reads the diskette. This will continue for several seconds while the computer loads the disk. When the disk drive stops, you will see a message similar to the following:

Current date is MON:01-08-83
Enter new date:

If instead of booting the disk, the computer returns a message to the screen which reads:

NonSystem disk or disk error
Replace and strike any key when ready

you should check to make sure you have the MS-DOS disk in Drive A (the one closest to your computer screen) and then press any key.

If you are sure that you are using the correct disk, and that it is inserted in Drive A, and repeated efforts at Warm and Cold Starts do not load the software, return the MS-DOS disk to your Chameleon dealer for further investigation.

SETTING THE DATE

Once the system has properly booted the MS-DOS disk you will be asked to provide date and time information for the operating system load routines. The first line on your screen (illustrated previously) shows the format of the date expected. It is read, day of the week, month-day-year. The second line is prompting you to enter the current date. If the date shown is the date you want to use simply hit the Enter key, if it is not, enter the current date.

In order to be accepted as valid, the date must be formatted as follows:

- Key in the current month as either one or two numeric digits between 1 and 12.
- Key in either a hyphen (-) or a slash (/) as a separator.
- Key in the day as one or two numeric digits between 1 and 31.
- Again, key in a – or / as a separator, using the same symbol you used between the month and day.
- Key in either the last two digits of the current year between 80 and 99 or four digits between 1980 and 2099.

• Finally, hit the Enter key to send the information to MS-DOS.

If you have keyed and entered a date in the required format, MS-DOS will proceed to prompt you for the current time. If the date format is unacceptable it will respond with:

Invalid date
Enter new date:

At this point you can reenter the date and hit the Enter key. MS-DOS will accept a number of formats for a given date. For example for the date January 8, 1983, it will accept:

1-8-83
1-08-83
01-8-83
01-08-83
1/8/83
1/08/83
01/8/83
01/08/83

However, it will not accept:

1 8 83 because blanks are not allowed
Jan-8-83 because letters are not allowed
010883 because there are no separators
13-08-83 because the month is too big.

The list of unacceptable dates is infinite.

SETTING THE TIME

Once the date has been accepted, MS-DOS will prompt you for the time with a message similar to the following:

Current time is 11:50:41.75
Enter new time:

The above time display would be read--11 hours:50 minutes:41 seconds.75 hundredths of a second.

To respond with the correct format you must:

• key one or two digits for the hour; the hour must be less than 24,

- key a colon (:),
- key one or two digits for the minutes; the minutes must be less than 60.

The seconds and hundredths of seconds are optional. If they are not keyed zero will be used. If you want to enter seconds continue with the following:

- key a colon (:)
- key one or two digits for the seconds; the seconds must be less than 60.

If you also want to enter hundredths of seconds:

- key a period (.),
- then key one or two digits between 0 and 99.

Once you have keyed the time, press the Enter key to send the information to MS-DOS.

If the format is incorrect MS-DOS will respond with:

Invalid time
Enter new time:

If this occurs re-key and re-enter the current time taking care to follow the format described above.

Remember, the format is:

hours:minutes:seconds.hundredths

and only numeric digits are acceptable. Review the time keyed carefully before using the Enter key.

It is possible to override the preceding date/time formatting by simply booting the MS-DOS disk and pressing the Enter key twice in quick succession when you are asked to enter the date. However, it is wise to get into the habit of good programming techniques early on. The date/time entered correctly now will be reflected on your program and text files when calling up the directory for the disk. This will in turn help you remember when a change or new entry was made.

After the time is accepted, MS-DOS will take a few seconds to finish initializing memory. When it has finished, a greeting similar to the following will be displayed:

MS-DOS Version 1.1 February 12, 1983
A:

The Chameleon is now ready to operate under the control of MS-DOS. Notice, in particular, the ***A:*** on the last line. This is the MS-DOS prompt. It serves as a signal to you that MS-DOS:

- is in control,
- is waiting for a command, and
- that disk drive A is the default drive.

When using MS-DOS, you will also see ***B:*** used as a prompt. This signals similarly that MS-DOS:

- is in control
- is waiting for a command, and
- that disk drive B is the default drive.

Note: ***When disk drive A is the default, any MS-DOS command that results in data being read from or written to a disk will use disk drive A unless the drive is referenced specifically. Likewise, when disk drive B is the default, drive B will be used.***

You can change the default drive from A to B by simply keying **b:** after the A: and pressing the Enter key.

Try that now, and you will see the prompt change from A: to B:.

To change the default back to drive A, key **a:** after the B: and press Enter.

THE KEYBOARD UNDER MS-DOS

MS-DOS assigns particular meanings to certain keys on the keyboard. The following section gives a description of the keys that perform special functions when MS-DOS is in operation.

- The insert key, ***Ins***, is found below ***PrtSc*** and ***1*** on the numeric pad. This is used to insert characters within a line.
- The delete key, ***Del***, is to the right of the Ins key. It is used to erase the character to the left of the cursor.

The following function keys on the left side of the keyboard are used by MS-DOS when data is being edited with the line editor.

- F1 - Displays previously entered information one character at a time.
- F2 - This key is used with another character. First press ***F2***, then key another character, and MS-DOS will display the previously entered line up to the character key entered after the ***F2***.

- F3 - This displays all of the last line key-entered.
- F4 - This key is also used with another key. First press ***F4***, then key another character; MS-DOS will then display the part of the previously entered line that follows the character keyed.
- F5 - This will save the line currently on the screen for further editing.

Now that you are familiar with the keyboard, you are ready to continue using the MS-DOS disk provided with the Chameleon. Make sure it is still inserted in disk drive A. Boot the disk, either by turning the Chameleon on, or by resetting with the ***Ctrl, Alt and Del*** keys.

Since the MS-DOS disk is in drive A, we will want the default drive to be drive A in the following exercise. Check the screen for the prompt A:. If B: is displayed, key:

a:

and then press the Enter key.

COPYING A DISK

The MS-DOS diskettes are extremely important to efficient operation of the Chameleon--so in the following exercise you will make back up copies of these diskettes to be used on a daily basis. Once the backups have been made, the original MS-DOS diskettes should be put away in a protected place. It is a wise practice to make copies of all your important diskettes and use the copies so the originals can be protected from the hazards of frequent use.

Since the Chameleon comes standard with two disk drives, copying a diskette is a simple matter. In this important exercise you will make backup copies of the MS-DOS diskettes provided with the Chameleon. Since the backups you are making will be used frequently in your day to day computer work, it is preferable that you use new diskettes for the backups. However, it is not mandatory. The copy operation will erase any information that was on the destination diskette and replace it with the information from the source diskette that is being copied. Make sure you use a diskette that does not contain information you will want later.

At this point you should have the first MS-DOS diskette in drive A. Place the diskette you are going to use for a backup in drive B. At the A: prompt, key in:

diskcopy ab ⟨CR⟩

Note: *⟨CR⟩ means to press the Enter key.*

MS-DOS will display the following message:

Chameleon Computer Disk copier vs. 1.00
Put source disk in drive A
Put destination disk in drive B
Strike any key

Note: ***If you were going to copy a source diskette other than the MS-DOS diskette, you would remove the MS-DOS diskette andreplace it with the source diskette at this time.***

Since you have already inserted the proper source and destination diskettes in the appropriate drives, simply strike any key. Any single key can be used; however, it is good to get into the habit of responding to this prompt with a particular key for ease in retracing your steps if something should go wrong. Choose a key now that is convenient to key--such as the space bar or the letter "j" and get into the habit of being consistent.

Once you signal that you are ready by striking any key, you will see the message:

Formatting destination disk .....

and you will see and hear drive B in operation. Formatting is required on soft-sectored disks because the operating system needs to know how to reference data on a disk. Without this knowledge of the location of individual records of data, the disk drive would not be of any use as a storage device. Tracks and sectors are mapped out during a format operation and the disk ***directory*** is initialized and cleared. These steps will allow the operating system to reference every track and sector properly to find or write any record.

When the formatting portion has been completed, the screen will display:

Copying .....

signalling that the copy operation is now taking place. After each file is transferred from drive A to drive B, a file verify is performed to make sure the copy operation was successful. When the copy is finished, the following message is displayed:

Copy another disk (Y)es or (N)o

If you want to make another copy of this disk, place a new diskette in Drive B and key:

y

Since there is another MS-DOS disk that you wish to make a backup copy of, remove both disks from both drives, and place an appropriately marked sticky label on the copy you just made. Use a ***felt tip*** pen, put a creation date as well as a description on the label, and be sure to mark the label before you place it on the disk.

Put the second MS-DOS disk in drive A, put a blank disk in drive B and key:

y

MS-DOS will then repeat the sequence previously described. If you are finished making copies, key:

n

The A: prompt will appear signalling that MS-DOS is waiting for another command.

A word of caution concerning disk copying. Most software companies are kind enough to make their diskettes 'copy-able'. This in no way means that you should be making unlimited copies of canned software and distributing them to friends. This action is termed ***pirating*** and is not viewed kindly by the software companies who, as a rule, will prosecute violators to the full extent of the law.

COMPARING TWO DISKS

The DISKCOMP utility program is used to compare two disks to see if they have identical or dissimilar data on a track-by-track basis. As stated before, a file verify operation is performed on each file as part of the DISKCOPY operation. This makes comparing the two disk unnecessary. But there are times when you may need to compare two disks without a copy being performed.

To perform the DISKCOMP utility, place the first MS-DOS disk in drive A and Enter the following:

diskcomp ab ⟨CR⟩

What you have told the operating system to do is load and run the disk compare utility on the two disks located in drives A and B. MS-DOS will now display the following message:

Chameleon Computer Disk compare utility vs. 1.00
Put first disk in drive A
Put second disk in drive B
Strike any key to continue

At this point you would remove the first MS-DOS disk and place the two disks you wish to compare in the disk drives. Since you have just copied the MS-DOS disks, you will use the disk in drive A and the backup copy you made of it as the disk in drive B. Strike a single key. You will then see the following message:

Comparing 1 side(s)

and the two disk drives in use. When the comparison operation is finished, MS-DOS will display:

Diskettes compare ok
Compare more disks (Y)es or (N)o

If you want to compare more disks, you can key **y** and the sequence will be repeated. If you are finished, you can key **n** and the A: prompt will be displayed. For now don't key anything; just let the system wait while you organize the disks.

Now, just for practice, you will compare two different disks. Take any other disk provided with the Chameleon, replace it with the MS-DOS backup disk in drive B and Enter:

y ⟨CR⟩

Since the two disks are not the same, MSDOS will display the following message:

Compare error on track 00

which indicates that the differences were found on the disks starting in track 00. The program will continue to show all tracks which were not identical. Tracks are numbered 00 thru 39 for single-sided disks and continue on from 40 to 79 on double-sided disks.

When all the tracks have been scanned, the program will display the following message:

Disks are not the same
Compare more disks (Y)es or (N)o

If you are finished, key:

n ⟨CR⟩

This will end the DISKCOMP operation and return the A: prompt.

There are many MS-DOS commands and utility programs which perform other vital systems operations. To gain full understanding of these functions and to get full use out of your Chameleon, you should study the MS-DOS instruction manual provided with the Chameleon.

MS-DOS FILES

In addition to knowing the MS-DOS ***commands*** and what they do, it is also important to understand MS-DOS ***files***. MS-DOS organizes and stores information in files. Information can be entered and processed into the Chameleon and it can be printed without ever creating a file. ***However, if information is to be saved on a disk and used again, it must be contained in a file.***

There are many different kinds of files: data files, program files, command files, etc. Like information in the folders in a file cabinet, the information in a given disk file is usually related in some way. The size of files also varies from several characters to several disks of information. The person creating the files, determines the size and organization of a file.

The variety of files is infinite. Some typical data files might contain names and addresses for a mailing list. Another file might contain a form letter. Yet another file might contain a program to print the form letter for each of the names and addresses found in the first file.

Typically, files are created or read into the RAM memory of the Chameleon, and then read onto and stored on disks.

One single-sided disk can hold up to 64 distinct files. The total amount of information stored in from 1 to 64 files on a disk is approximately 160 K or approximately 160,000 characters of information. This is roughly equivalent to some 110 pages of type written material.

Each file on a disk has a unique name and location. Both are important. When you create a file and put it on a disk, you assign it a name. You also determine which disk it will go on by placing a particular disk in a particular drive and specifying that drive when you issue a command to store the contents of the file. An example of a fully qualified filename might be:

a:memo.jun

In this fully qualified name the ***a:*** is a drive specifier; ***memo*** is the primary filename and ***.jun*** is the extension.

The drive specifier indicates which drive is to read or write the file. In this example, the file named ***memo.jun*** would be placed on or read from a disk in drive A. If the same disk were placed in disk drive B, the fully qualified name would be:

b:memo.jun

The primary filename is 1 to 8 characters and is commonly used to describe the contents of the file. Every file must be labeled with at least a primary filename, containing at least 1 character.

The file extension is an optional qualifier. The extension always starts with a period and is 0 to 3 characters in length. The extension is often used to indicate what type of data is in the file, as opposed to the primary filename which is usually descriptive of the data content. Extentions are also used to group data files.

The fully qualified name is not always necessary to locate a file. The drive specifier can be omitted when the file is on the default drive. This means if the file is on a disk in drive A and the prompt is ***A:***, the ***a:*** does not need to be used. Likewise if the file is on drive B and B is the default the ***b:*** may be omitted. The file extension is also an optional part of the name. However, once a file has been created using a name with an extension, the extension must be used to locate the file.

When the file is created or stored, MS-DOS places the primary filename and, if there is one, the extension in a directory on the disk. The directory tracks exactly where the file is located on the disk. This directory and the specified or implied drive identifier are used to locate the file.

The information on the MS-DOS disk is also stored as disk files. You can see the list of files on your MS-DOS disk by using the directory command.

To do this make sure the MS-DOS disk is in drive A and that the default drive is A: i.e. that the *A:* prompt is on the screen and key:

DIR

then hit the Enter key.

You should see something similar to the following:

COMMAND	COM	4959	5-0782	12:00P
FORMAT	COM	5769	5-0782	12:00P
CHKDSK	COM	4879	5-0782	12:00P
SYS	COM	3579	5-0782	12:00P
SAMPLE	BAS			

This is a list of some of the files on the MS-DOS disk.

You could have also produced the same list by keying and entering:

dir a:

This format of the command is useful if the files you want to list are on a disk other than the current default drive.

Look again at the list of files on the MS-DOS disk. Often the extensions have particular meanings that help identify what kind of information is in the file. Most of the files on the MS-DOS disk end in either:

.com

or

.exe

The extensions *.com* and *.exe* indicate that the information is a command. Other extensions that have a particular meaning to MS-DOS are:

.bas	.bak
.lib	.hex
.obj	.map
.$$$	

For explanation on the meaning of each of these extensions, consult the MS-DOS manual.

To the right of the files should be the date/time signature which DOS has been keeping track of since you entered them after booting the DOS diskette. Whenever a file is written or saved to disk, DOS time-stamps the file as part of its housekeeping operations. If you did not enter the correct date and time, DOS will use the default date/time it displayed when you booted up--and your files will be stamped incorrectly.

ASSIGNING FILENAMES

There are a few rules to follow in assigning names to your files. The first and most important is never use the same name for two different files on the same disk. It won't work! When the second file is put on the disk, it will write over and erase the original file with the same name. If you are unsure about which names have been used already, use the directory command, ***dir***, to list the names of the existing files on a disk.

It is also important that the filenames be meaningful to you and other people that may be using them. Although the computer recognizes any unique combination of characters, people need more hints. For example the files described previously could be named:

address
letter19.jun
prntletr.bas

to indicate that the files contained your address list, the nineteenth letter sent out in June, and your standard letter printing program.

They could also be named:

123456789
@$^.ab
p.bas

Although either set of names is acceptable to the computer, the second set provides little information to you about the contents of the files.

Finally, remember the folowing MS-DOS rules:

- The main file name must be from 1 to 8 characters long.
- If an extension is used, it must contain a period followed by 0 to 3 characters.
- The characters used in the file name and extension can be:

  the letters of the alphabet
  the numbers 0 through 9
  the following special characters:

  ~ : { } ! @ # $
  % ^ & * ( ) – ( )

Note in particular that periods can only be used as the divider between the filename and extension, not as a character in the name.

Note also that a number of special characters cannot be used such as ***commas, asterisks, question marks, colons, slashes, etc.*** These characters cannot be used in filenames because they all have a special meaning or use in MS-DOS.

- The colon is used to indicate the preceding letter is a drive specifier.
- The period indicates that the following characters are part of the filename extension.
- The comma is used to separate filenames.
- The slash is used to indicate disk format descriptions.
- The asterisk and question mark are both used as ***global characters***.

GLOBAL CHARACTERS

Global characters are used to represent "any character" in filenames in MS-DOS commands. The question mark, **?**, is used to represent any character in a particular position. The asterisk, *****, can represent a group of characters.

Global characters are particularly useful in locating related files or in performing an operation on a number of related files with one command. This concept is easiest to explain with examples. Suppose you have a disk with the following files:

memolou.jun
memosue.jun
memolou .jul
memojon.aug
memosmit.jun
records.jun
records.jul

You could get a complete list by keying and entering:

dir

or

dir a:

You could see the name of a particular file by keying and entering one name such as:

dir a:memolou.jun

You could also use two global characters and see all memos by keying and entering:

dir memo*.*

You could also see all activity in June by keying and entering:

dir *.jun

You could also key and enter:

dir memo*.jun

This would result in a list showing all the memo files with an extension of '.jun'.

memolou.jun
memosue.jun
memosmit.jun

Question marks could also be used to get some of the same listings. To illustrate the difference between using a asterisk and using question marks compare the following example with the previous one.

A command of:

dir memo???.jun

would result in a list showing:

memolou.jun

and

memosue.jun

The filename memosmit.jun would not be listed because it contains 8 characters in the filename and only 7 positions were specified in the ***dir*** command.

Notice that when the asterisk was used instead of the three question marks, ***memosmit.jun*** was also included on the list. You can see from the previous examples that the more logical and consistent the filenames are, the better use you can make of global characters.

Global characters are extremely powerful, but they can also be dangerous particulary when you are erasing files. The command:

dir *.*

would list all files on the default.

Similarly, the command:

erase *.*

will erase all files on a disk.

The summary of commands in this Chapter have illustrated a few MS-DOS commands and conventions, for additional information consult your MS-DOS manual, and have fun experimenting on your own.

CHAPTER 6
GETTING STARTED WITH PERFECT WRITER

This chapter is written for the user who wishes to get the word processing program Perfect Writer up and running in a hurry. If you bought your Chameleon with the sole intention of using the system for the spreadsheet programs or as a means of creating programs in either BASIC or assembly language, you're in for a pleasant surprise.

LET'S GET GOING

We recommend that you take the time to go through the Lessons diskette that came with your Perfect Writer package. This will demonstrate all the magnificent features available to you. We will begin with a simple letter writing example here just to show you the benefits.

Place your working copy of DOS in Drive A, boot it through either a Warm or Cold Start, answer the date and time queries, and finally look for the A: prompt. If all this seems unfamiliar and you think we are speaking a foreign language, you should read *Chapter Two and Chapter Five* to get the basics down first. Operating a computer is simple, but you need a firm understanding of the ground rules before you move into the big league.

MAKING BACKUP COPIES

Now that you have the operating system in control, you will make backup working copies of the Perfect Writer disks. This is a fairly simple operation if you follow along closely.

Making backup copies of all software that is not copy protected will be a routine procedure you will always want to follow. A backup copy of an expensive disk assures you that even if the working copy gets erased accidentally, you will always have the original to make more working copies.

To make the backup copies of the necessary Perfect Writer files, you will need two blank diskettes. If you own a Chameleon Plus or are utilizing a double-sided disk drive, you will only need one blank. Place a blank disk in Drive B while the working copy of DOS is in Drive A. If the A: prompt is not on the screen, reboot the DOS disk. Now type the following

and press the Enter key:

format b:/1/s for single-sided drives

or

format b:/s for double-sided drives

After formatting has been completed on the first blank you will see the following prompt on the screen:

Format another (y/n)?

Remove the disk in Drive B, replace it with the second blank and answer the prompt with:

y

When the second disk has been formatted, answer **n** to the **Format another** prompt. Take the working DOS disk out of Drive A and replace it with the Perfect Writer disk marked **EDIT DISKETTE.**

At this point some of you may be asking yourselves why you shouldn't just use the **DISKCOPY** function of DOS to make your backups. A brief explanation is necessary.

The **DISKCOPY** function will reproduce all the files in the directory of the source diskette on the destination diskette. It does not put the operating system, which is necessary to boot the computer from a Cold Start, onto the destination diskette unless it is on the original diskette that you are copying.

It thereby creates a Slave disk which relies on a Master diskette to boot the system for it. You will find that putting the operating system on all your new diskettes is desirable. This will eliminate the need to swap disks just to boot the system. Most software packages are shipped on diskettes that do not contain an operating system.

If you've been following along, you should know that nothing can be accomplished on any computer until the operating system has been installed or booted into the computer's memory.

The **FORMAT** command that you just typed automatically puts the system on your destination diskette when you use the /s. It also puts on the ***COMMAND.COM*** file which is part of the operating system. Copying ***COMMAND.COM*** alone will not do the job. It must be installed at format time.

For those desiring more detailed explanation, refer to the **SYS** and **FORMAT** commands in the DOS manual that came with your system.

Type the following at the A: prompt and press Enter:

copy *.* b:

One by one, each of the files on the **EDIT DISKETTE** will be copied to the blank disk in Drive B. When this procedure is finished, the operating system should show that five (5) files have been copied. They are:

MENU.COM PW.COM PF.COM PF.DAT PW.HLP

If for any reason these files are not on the disk, check the directory of files on **EDIT DISKETTE**. If any or all are missing, use the **DIR** command to find the files just listed on one of the other Perfect Writer diskettes and copy them to your first blank disk.

These files will allow you to use Perfect Writer, Perfect Formatter, the MENU, and the HELP file without switching disks all the time. You will be able to print off copies in a simple manner described in your manual as ***FORMATTING FOR 'VANILLA' PRINTER***. This will be your main controlling disk when using the Perfect Writer word processing package.

Remove the disk from Drive B, get out a felt tip pen and locate the sticky disk label sheet that came with your blank diskettes. Mark one of the labels **PERFECT WRITER EDIT DISKETTE** or any other description that will help you to remember the contents of this disk. It would also be a good idea to list the six files that are on the disk.

Remove the label from the sheet and carefully place it on the disk right next to the manufacturers label. Don't cover the write-protect notch with the label.

You are now ready to make your second working disk. Place the blank disk you formatted first in Drive B. Drive A should still have the **EDIT DISKETTE** in it and the A: prompt should be on the screen. Type the following:

copy menu.com b:

One (1) file should have been copied to the second blank disk.

Remove the **EDIT DISKETTE** from Drive A and replace it with the **INSTALLATIONS DISKETTE** from the Perfect Writer package. Type:

copy *.com b:

Three (3) files should have been copied this time. Now type:

copy pf.dat b:

and One (1) file should have been copied.

Enter:

dir b:

and the following files should be on the directory of the disk in Drive B:

COMMAND.COM	MENU.COM	PFCONFIG.COM
PF.COM	PF.DAT	PP.COM

As before, if any or all of these files do not appear on the B directory, check the Edit Diskette and Installations Diskette to make sure they are on the original disks; and if not, copy the listed files from one of the other diskettes that came with the Perfect Writer package.

You have now completed the backup operation on both disks.

Although this second disk will not allow you to edit text files, it will permit you to format a text file for printers which you will define in the PFCONFIG.COM file. This, in turn, will install the PF.DAT. Perfect Formatter will use the PF.DAT file to configure the text file correctly on the default printer you choose.

Remove the **INSTALLATIONS DISKETTE** from Drive A and put it in a safe place with the other diskettes in the Perfect Writer package. Mark another sticky label: **PERFECT WRITER INSTALLATIONS DISKETTE** and place it on your second working diskette when you remove it from Drive B.

USING THE WORD PROCESSOR

Place the working copy of Perfect Writer Edit Diskette in Drive A. There is enough space on this diskette for the example, but it is always a good idea to put your text files on another disk in Drive B while the word processor controlling program diskette is in Drive A.

With the A: prompt on the screen Enter:

menu

At this point the operating system is loading the menu.com file into memory and executing its instructions. You should see the following after a few seconds:

Selections:

C reate a document
F ormat
P rint a document
R evise a document
S pelling check
D irectory of a disk
E rase a document
Q uit to DOS

Type one character to indicate your selection now:

The best way to learn the MENU is to play with it.

Typing **d** will cause the MENU to ask for the desired drive. Type **a** for the the A drive and **b** for the B drive. It will go out to the directory of the selected drive and show you what is on that diskette. Pushing any key will return you to the MENU. See what's on the directory for both drives. Type in a drive that isn't there (like ***c***) and see what happens.

If you happened to purchase Perfect Speller and type an **s**, the MENU will ask for the document you wish to check, and proceed to look up all the words of your document in its dictionary. You can disregard misspelled words, add them to the dictionary, create a special dictionary, or have them marked in the document to correct afterward. Perfect Speller is very fast and very powerful.

If you do not have Perfect Speller, selecting **s** on the menu will just give you a message that the spelling checker program could not be found and lead you back to the MENU.

Typing **q** will get you out of Perfect Writer MENU and back to the operating system. The A: prompt will appear when you select **q** if that is the default drive.

Since you have no documents to Format, Print, Erase or Revise, you should now select **c** from the MENU and Create a document. The MENU will prompt you to name the file you wish to create. If you just press the Enter key without naming any file, Perfect Writer will create **NAME.ME** and use the default drive to produce a file name.

You can call a document any descriptive name you desire, but for the purposes of our demonstration, call your file **example.mss**.

When the screen appears, the bottom line will be highlighted and a lot of information will be visible. The version of Perfect Writer is first displayed followed by (***WRAP***) meaning the word wrapping function is active. The word ***example*** is next. This is the name of the buffer that the text file occupies in memory at the moment. The name of the file you called up comes after the buffer. Notice how the buffer name is the same as the file name. And last a *-0%-* which tells how much of the text file has been edited.

You're now ready to enter the example so begin by typing a thank you note to dear Aunt Martha.

Type the following:

Dear Aunt Martha:

and hit the Enter key two times. If you like to indent your paragraphs, press the Space Bar at the bottom of the keyboard three times. Now finish typing the body of the letter:

Thank you very much for a wonderful party the other night. I can't remember when I've had such a great time, or had so much good food set on the table. You really know how to do these things right.

Hoping to see you again real soon. Say hello to Uncle Jack.

Yours truly,

SAVING THE TEXT FILE

Now that you've finished keying in the example, it would be appropriate to save a copy of it to disk. Using the extended command ***Ctrl--x, Ctrl--s*** will do this for you. Hold one finger on the ***Ctrl*** key. With another finger, first press the ***x*** key then the ***s*** key and let go of the ***Ctrl*** key. You have just performed an extended command (Ctrl--x) that saved the document (Ctrl--s) to disk.

Notice that the letter is still on the screen ready for more input from you. If you happened to accidentally erase or lose the screen image now, you could easily call up the file from the disk again. Getting into the habit of performing this save routine after every few entries is the best thing

you can do for yourself. You may lose a little of your work if you do have an accident, but that's certainly better than having to re-key the whole thing again from scratch.

If you have saved your document to disk, take a look at the numeric pad. Press the **7** key marked ***Home***. Notice that the cursor has moved to the top of the document. If your document were many pages long, it would have taken you to this same spot. This spot is known as the beginning of the file.

Now press the **1** key marked ***End***. This should position the cursor at the bottom of the letter. Even if your letter were 25 pages long and you were positioned at the very top, the End key would bring you to the very end of the document. This spot is known as the end of the file.

Experiment with the arrow keys on the numeric pad to move the cursor around the screen. Use the Backspace key to delete a letter. Use the Enter key to add or insert extra lines. If you add enough lines so that your letter is now bigger than just the one page you started on, use the ***Pg Up*** and ***Pg Dn*** keys to scroll up and down a page at a time.

Pretend that you really are going to send this letter to Aunt Martha. You're probably saying that what you have on the screen really is a mess and you wish you had the original letter. No problem! Type the extended command for quitting the edit session **Ctrl--x, Ctrl--c**. The Quit command will allow you to make a software-controlled exit from Perfect Writer. (This type of exit is always the best way for you to leave any program.)

After a few seconds you should be back at the MENU. Select the Revise a document option by typing **r**. When the MENU prompts you for the name of the file, type:

example.mss

If you have been storing the file on the B drive make sure you precede the name with ***b:***; otherwise it will look for the file on the A drive and not find it. It would then think you are trying to create a new file and give you a blank overlay to type your letter on.

The MENU should now be looking for the file you saved awhile ago. In a short time, it will display the original letter on the screen without all the extra spacing, erasures and odd letters you added when you were fooling around with the cursor control keys.

PRINTING THE LETTER

Printing off a hardcopy of the letter is as simple as saving it to disk. Make sure you have followed the instructions in *Chapter 4-Setting Up Your Chameleon* for connecting your printer to the correct output port on the back of the Chameleon. Make sure there is paper in the feeding mechanism and that the printer is turned on.

Type **Ctrl--x, Ctrl--p**, the extended command to print the buffer exactly as it appears on the screen. Within a few seconds the printer should kick over and churn out the letter.

Finally, you will want to exit from the program. Again, use the Quit command, **Ctrl--x, Ctrl--c**.

The example was quite a simple one and did not begin to use all the commands available in Perfect Writer. Yet you can begin to see the power you now have over your word processing problems.

HOW TO USE THE LESSONS DISKETTE

The people at Perfect Software have gone to a lot of trouble to make learning Perfect Writer as easy as possible for you. Just put your working copy of Perfect Writer in Drive A and the LESSONS DISKETTE in Drive B. Call up the MENU and begin by selecting r for Revising a document. At the prompt, type in:

b:teachme.pw

This is your first introduction to the lessons and how to call them up. Among the subjects discussed are the cursor, keys, extended commands, how to cancel any command, use and definition of all command and function keys, seeing the directory, quitting an edit session and reading in a file from the disk.

The lessons run as follows:

Lesson 1 - Creating a document
Lesson 2 - Revising a document by inserting and deleting
Lesson 3 - Moving the Cursor
Lesson 4 - Moving and Copying Text
Lesson 5 - Searching
Lesson 6 - Using Multiple Windows
Lesson 7 - Formatting

After a few brief lessons, you'll be turning out professional looking letters, documents, memos, term papers and all the things you would have normally done by hand or on a conventional typewriter. The Document Design section of the Perfect Writer manual will show you how to turn out any type text handling chore you can think of. You can even write your next novel with it.

OTHER CHAMELEON CUSTOMIZED FEATURES

Although Perfect Writer has been configured to the Chameleon for you, certain keys may not work exactly as described in the Perfect Writer manual, the Lessons diskettes, or on the quick reference cards provided by Perfect Software.

A Read-Me-First document may have come with your Perfect Writer package explaining some of the substitutions and a few features unique to the Chameleon. We will now look at a few of these and several others that are not in the document.

FUNCTION KEYS

Perfect Software has implemented the ten functions keys on the left hand side of the Chameleon keyboard in such a way that nearly 40 of the Perfect Writer commands are produced. This is accomplished by utilizing a four-level access method in the following manner:

Function - *Command level one. These commands are executed by simply pressing the desired Function key.*

Shift-Function - *Command level two. These commands are executed by holding down the shift key and pressing the desired function key.*

Ctrl-Function - *Command level three. Execute these commands by holding down the 'Ctrl' key and pressing the desired function key.*

Alt-Function - *Command level four. Hold the 'Alt' key and press the desired function key.*

Following are the Perfect Writer commands implemented by the function keys at each level:

FUNCTION COMMAND KEYS

F1 (HELP)	Display Function key commands
w/Shift	Display Shift-Function commands
w/Ctrl	Display Ctrl-Function commands
w/Alt	Display Alt-Function commands

F2 (Buffer)	Switch Buffer	F6 (Search)	Search Forward
w/Shift	Buffer Directory	w/Shift	Search Backward
w/Ctrl	Delete Buffer	w/Ctrl	Query Replace
w/Alt	N/A	w/Alt	Global Replace
F3 (Copy)	Set Mark	F7 (Input)	Find Document
w/Shift	Yankback	w/Shift	Read Document
w/Ctrl	Copy Region	w/Ctrl	Insert Document
w/Alt	Delete Region	w/Alt	Disk Directory
F4 (Window)	Two Windows	F8 (Output)	Save Document
w/Shift	Other Window	w/Shift	Write Document
w/Ctrl	One Window	w/Ctrl	Write Region
w/Alt	N/A	w/Alt	N/A
F5 (Display)	Adjust Paragraph	F9 (RESERVED)	
w/Shift	Center Line		
w/Ctrl	Set Right Margin	F10 (Special)	Exit DOS
w/Alt	Set Left Margin	w/Shift	Repeat Next Command
		w/Ctrl	Check Spelling
		w/Alt	Where Am I?

Each of these commands can also be executed using the regular ***Escape-Control*** command structure of Perfect Writer. Use the function keys if you find them convenient and helpful.

SPECIAL KEYS

The following keys will perform the described commands:

KEYS	COMMAND
Ctrl--x. (period)	Set Left Margin
Ctrl--xf	Set Right Margin
Ctrl--x(tab key)	Set Tab Spacing
Esc--@	Force Page Break
Esc--?	Directory of Commands
Esc--Esc	Display Alt-Function Commands

Also, the following numeric pad commands:

Esc--Del Right justifies all characters in line--no erasing.

Esc--1 (End) Inserts a line--same as Ctrl--o.

Esc--2 (▼) Beginning of paragraph--same as Esc--p.

Esc--3 (Pg Dn) Close Insert--same as Esc, Ctrl--o.

Esc--4 (◀) Deletes sentence forward--same as Esc--k.

Esc--8 (▲) Beginning of paragraph you are in only--no correlation to other commands.

On the quick reference card under the **DELETE AND INSERT** commands, the **DEL** key will not work as described. On the Chameleon, it works similar to the ***Ctrl--k***, deleting characters to the end of the line. Simply substitute the Backspace key everywhere the DEL key appears on the card to get the desired results.

We encourage you to take the time to read the enclosed Perfect Writer manual and go through the Lessons Diskette to discover the full extent of your new found 'tool'. You will be glad you did.

CHAPTER 7
GETTING STARTED WITH PERFECT CALC

This chapter is written for the user who wishes to get the spreadsheet program Perfect Calc up and running in a hurry. If you bought your Chameleon with the sole intention of using the system as a word processor or as a means of creating programs in either BASIC or assembly language, you're in for a pleasant surprise.

Perfect Calc is by far one of the most powerful and easiest to use spreadsheet programs on the market today. You may be asking yourself at this point, "What is a spreadsheet program and why on earth do I need it?". In very simple terms, a spreadsheet program is a method of entering financial or statistical data without the bother of recalculating totals, mathematical formulas, percentages, etc., over and over again. Your checkbook is a perfect example. You make a deposit and add the amount to what you already have in the bank. You write a check and subtract the amount from what you have in the bank. All the while you're keeping a running total of how much money is in your account.

By using Perfect Calc, you can keep better track of your bank account and where the money goes. You will be using a check book balancing example to run through Perfect Calc in this chapter, but there are hundreds of other uses both great and small.

If you trade on the market and own stocks or bonds, a spreadsheet can help you keep tabs on how well you or your broker is handling your investments. And after you've worked with Perfect Calc awhile, you may be able to predict what stocks will or won't do and when to buy or sell.

If you're a sports buff, Perfect Calc can keep you posted on how well your favorite team is fairing against other teams in the league. You may even be able to predict the outcome of the Super Bowl or World Series --or even your home town bowling league.

"But I'm a housewife! How is a spreadsheet program going to help me?". A great deal! You may have a favorite recipe whose ingredients are proportioned for four servings. Wouldn't it be terrific to have the proportion of ingredients recalculated for two? Or how about when you're going to throw a party and need to know how much to buy for twenty guests?

These are just a few of the ways a spreadsheet program can be used. You probably came up with several others while you were reading the ones just given, and you'll be coming up with dozens more once you get the hang of it.

LET'S GET GOING

We recommend that you take the time to go through the Lessons diskette that came with your Perfect Calc package. This will demonstrate all the magnificent features available to you. We will however begin with a check book balancing example just to show you the benefits. Place your working copy of DOS in Drive A, boot it through either a Warm or Cold Start, answer the date and time queries, and finally look for the A: prompt. If all this seems unfamiliar and you think we are speaking a foreign language, you should read *Chapter Two and Chapter Five* to get the basics down first. Operating a computer is simple, but you need a firm understanding of the ground rules before you can move into the big league.

MAKING BACKUP COPIES

Now that you have the operating system in control, you will make a backup working copy of the Perfect Calc disk. This is a fairly simple operation if you follow along closely.

Making backup copies of all software that is not copy protected will be a routine procedure you will always want to follow. A backup copy of an expensive disk assures you that even if the working copy gets erased accidentally, you will always have the original to make more working copies.

To make the backup copy of the necessary Perfect Calc files, you will need a blank diskette. Place the blank disk in Drive B while the working copy of DOS is in Drive A. If the A: prompt is not on the screen, reboot the DOS disk. Now type the following and press the Enter key:

format b:/1/s for single-sided drives

or

format b:/s for double-sided drives

After formatting has been completed answer **n** to the **Format another (y/n)?** prompt. Take the working DOS disk out of Drive A and replace it with the Perfect Calc disk. The other disk that came in the Perfect Calc

package is marked **LESSONS DISKETTE** and you will not be needing a backup of it. At this point some of you may be asking yourselves why you shouldn't just use the **DISKCOPY** function of DOS to make your backups. A brief explanation is necessary.

The **DISKCOPY** function will reproduce all the files in the directory of the source diskette on the destination diskette. It does not put the operating system, which is necessary to boot the computer from a Cold Start, onto the destination diskette unless it is on the original diskette that you are copying.

It thereby creates a Slave disk which relies on a Master diskette to boot the system for it. You will find that putting the operating system on all your new diskettes is desirable. This will eliminate the need to swap disks just to boot the system. Most software packages are shipped on diskettes that do not contain an operating system.

If you've been following along, you should know that nothing can be accomplished on any computer until the operating system has been installed or booted into the computer's memory.

The **FORMAT** command that you just typed automatically puts the system on your destination diskette when you use the **/s**. It also puts on the **COMMAND.COM** file which is part of the operating system. Copying **COMMAND.COM** alone will not do the job. It must be installed at format time.

For those desiring more detailed explanation, refer to the **SYS** and **FORMAT** commands in the DOS manual that came with your system.

Type the following at the A: prompt and press Enter:

copy *.* b:

One by one, each of the files on the diskette will be copied to the blank disk in Drive B. When this procedure is finished, the operating system should show that three (3) files have been copied. They are:

PC.OVL PC.COM PCCONFIG.COM

If for any reason these files are not on the disk, check the directory of files on Drive A. If any or all are missing, you will have to return the Perfect Calc package to the dealer where you purchased your Chameleon for further investigation.

Remove the disk from Drive B, get out a felt tip pen and locate the sticky disk label sheet that came with your blank diskettes. Mark one of the labels **PERFECT CALC WORKING DISKETTE** or any other descriptive title that will help you to remember what is on this disk. It would also be a good idea to list the four files that are on the disk so you won't be wondering all the time.

Now remove the label from the sheet and carefully place it on the disk right next to the manufacturers label. Don't cover the write-protect notch with the label.

You have now completed the backup operation on the Perfect Calc disk.

Remove the Perfect Calc diskette from Drive A and put it away in a safe place with the other diskette in the Perfect Calc package.

Place the Perfect Calc working diskette in Drive A. There is enough space on this diskette for this example, but it is always a good idea to put your spreadsheets on another disk in Drive B while the spreadsheet controlling program disk is in Drive A.

Type:

pc check.mss

Press the Enter key. Drive A should be whirring and its ***LED*** should be lit.

Once all the excitement has died down, you should see a highlighted line along the top with what appears to be an alphabet and numbers running down the left side. The words ***NEW FILE*** should be in the lower left of the screen and ***check a1*** in the lower right.

Using the arrows keys located on the numeric pad (under the numbers 8 (↑), 4 (←), 6 (→) and 2 (↓)), move the highlighted cursor around the spreadsheet. Notice how the screen ***wraps around*** when you go beyond the screen boundaries. The spreadsheet is a whole lot bigger than just the section you see on your screen.

The alphabet at the top identifies the ***column*** your cursor is in and the numbers along the left side identify the ***row***. Wherever the cursor is, it will always intersect with a column and row. You started in ***a1***. The screen reflected this position (sometimes referred to as an ***element*** or ***cell***) in the lower right of the screen; whenever you move the cursor, this marking will change accordingly and let you know where you are.

There are 52 columns across (a-z, and again A-Z capitalized) and 255 rows down. To get back to where you started, press either one of the Shift keys and the) key at the same time. This is the **GOTO** command (lower left of the screen). Type **a1** and press the Enter key. The cursor should bring you back to the first position of the spreadsheet. Now that you know how to move around the screen, enter the example.

With the highlighted cursor in ***a1***, press the Space Bar. The word **LABEL:** should appear in the lower left of the screen. Type **my bank account** and press the Enter key. Notice that what you typed has now been transferred up to the a1 and b1 space. If you don't like this title or you'd like all the words in upper case for emphasis, press the Space Bar again, use the ***Ctrl--d*** keys to erase, change it to what you want, and press the Enter key.

If you begin to modify a cell and change your mind, you can use the Cancel Entry command ***Ctrl--g*** any time before you hit the Enter key. This will make the label, formula or numeric entry what it was before you started to change it.

Move the cursor over to ***d1*** with either the arrow keys or the GOTO command (***Shft--***)), press the Space Bar, type **DEPOSITS** and press Enter.

Move the cursor to ***f1***, press the Space bar, type **WITHDRAWALS** and press Enter.

Move the cursor down to position ***a3***. Press the Space Bar. Type **STARTING AMOUNT** and press Enter.

Move the cursor to ***d3***, type **1000** from the top row of the keyboard and press the Enter key. Congratulations. You just gave yourself $1,000.00.

Move the cursor to position ***a7***, and Enter the Label: **SUB TOTALS**.

Move the cursor to ***d7***. To add up all the DEPOSITS, press the = key. The words **Formula d7:** should appear in the lower left of the screen. Type **sum(d3:d6)** and press return. In an instant, **1000.00** should appear in position ***d7***. Position ***d7*** will add up everything you place in ***d3, d4, d5 and d6*** automatically from now on.

You will now want to do the same for WITHDRAWALS. Move the cursor to ***f7***, press =, type **sum(f3:f6)** and press Enter. You should see zeros in ***f7*** because you haven't entered any withdrawals yet.

Move the cursor up to *f4*, type **50** and press Enter. You have just indicated a withdrawal of $50.00 from your account. Position *f7* should have reflected this transaction.

You know, of course, this isn't all that needs to be done.

Move the cursor to *a9* and Enter the Label: **BALANCE**.

Move the cursor to *e9*, press **=**, type **d7 – f7** and press Enter. Position *e9* should hold the value 950.00.

To save your new bank balance spreadsheet, hold down the *Ctrl* key with one finger. With another finger, first press the *x* key then the *s* key and let go of the *Ctrl* key. You have just performed an extended command (Ctrl--x) that saved the document (Ctrl--s) to disk.

Notice that your spreadsheet is still on the screen ready for more input. If you happened to accidentally erase or lose the screen image now, you could easily call up the file from the disk again using the Read File command *Ctrl--x, Ctrl--r* or the Find File command *Ctrl--x, Ctrl--f*. Getting into the habit of performing this save routine after every few entries is the best thing you can do for yourself. You may lose a little if you do have an accident, but that's certainly better than having to re-key the whole thing again from scratch.

You might want to make a hard copy of your spreadsheet on a printer. This is easily accomplished by having followed the setup procedures described in *Chapter 4*, turning on your printer, and using the extended command *Ctrl--x, Ctrl--p*. As before, hold down the *Ctrl* key with one finger and press the *x* key then the *p* key and let up on the *Ctrl* key.

Finally, you will want to exit from the spreadsheet program. The Quit command, *Ctrl--x, Ctrl--c*, will allow you to make a software-controlled exit (which is always the best way for you to leave any program).

This example was a simple one and did not begin to use all the commands available in Perfect Calc. Yet you can begin to see the power you now have over your decision making problems.

HOW TO USE THE LESSONS DISKETTE

The people at Perfect Software have gone to a lot of trouble to make learning Perfect Calc as easy as possible for you. Just put your working copy of Perfect Calc in Drive A and the **LESSONS DISKETTE** in Drive B. Enter **pc** after the A: prompt. When the spreadsheet form is displayed on the screen, type the following:

Ctrl-x, Ctrl--r

to read in a file, then ask for the following at the **File to Read** prompt:

b:teachme.pc

This is your first introduction to the lessons and how to call them up. Among the subjects discussed are the cursor, keys, extended commands, how to cancel any command, use and definition of all command and function keys, seeing the directory, quitting an edit session and reading in a file from the disk.

The lessons run as follows:

Lesson 1 - Moving Around the Spreadsheet
Lesson 2 - Entering Data
Lesson 3 - Storing & Printing Your Spreadsheet
Lesson 4 - Replicating formulas, Inserting, Deleting and Moving
Lesson 5 - Copying Between Spreadsheets
Lesson 6 - Setting Up the Spreadsheet As You Like It
Lesson 7 - Using Associated Files
Lesson 8 - Concluding Suggestions

After a few brief lessons, you will be able to greatly expand upon your bank account spreadsheet. Some of the things you can add are: a column for check numbers, symbols for deposits of cash or checks, descriptive fields for transactions, columns which automatically keep track of utility, phone, newspaper, business, pleasure or any other type transactions.

We encourage you to take the time to read the enclosed Perfect Calc manual and go through the Lessons Diskette to discover the full extent of your new found tool. You will be glad you did.

FUNCTION KEYS

Perfect Software has implemented the ten functions keys on the left hand side of the Chameleon keyboard in such a way that nearly 40 of the Perfect Calc commands are produced. This is accomplished by utilizing a four-level access method in the following manner:

Function - *Command level on. These commands are executed by simply pressing the desired Function key.*

Shift-Function - *Command level two. These commands are executed by holding down the shift key and pressing the desired function key.*

Ctrl-Function - *Command level three. Execute these commands by holding down the 'Ctrl' key and pressing the desired function key.*

Alt-Function - *Command level four. Hold the 'Alt' key and press the desired function key.*

Following are the Perfect Calc commands implemented by the function keys at each level:

FUNCTION COMMAND KEYS

F1 (HELP)	**Display Function key commands**		
w/Shift	**Display Shift-Function commands**		
w/Ctrl	**Display Ctrl-Function commands**		
w/Alt	**Display Alt-Function commands**		
F2 (Buffer)	**Switch Buffer**	**F7 (Input)**	**Find File**
w/Shift	**Buffer Directory**	**w/Shift**	**Read File**
w/Ctrl	**Delete Buffer**	**w/Ctrl**	**Associate File**
w/Alt	**Delete Region**	**w/Alt**	**N/A**
F3 (Copy)	**Set Mark**	**F8 (Output)**	**Save File**
w/Shift	**Delete Element**	**w/Shift**	**Write File**
w/Ctrl	**Copy Region**	**w/Ctrl**	**Print Entire Spreadsheet**
w/Alt	**Copy Element**	**w/Alt**	**Print Region**
F4 (Window)	**Two Windows**	**F9 (RESERVED)**	
w/Shift	**Other Window**		
w/Ctrl	**One Window**	**F10 (Special)**	**Exit DOS**
w/Alt	**Fix Titles**	**w/Shift**	**Repeat**
		w/Ctrl	**Kill Entire Memory**
F5 (Display)	**Display Format**	**w/Alt**	**Where Am I?**
w/Shift	**Column Width**		
w/Ctrl	**Recalculation Mode**		
w/Alt	**Justification**		
F6 (Element)	**Replicate Formula**		
w/Shift	**Replicate Region**		
w/Ctrl	**Search Next Unlocked Element**		
w/Alt	**Edit Element**		

Each of these commands can also be executed using the regular *Escape-Control* command structure of Perfect Calc. Use the function keys if you find them convenient and helpful.

In addition to the Function keys, there are some special, dual-function keys with unique features to the Chameleon. These keys perform mostly *Ctrl* functions that would normally require two keys to be pressed. The special keys are located within the numeric pad and active when in normal, non-numeric operation. Engaging the ***Num Lock*** key will allow these keys to revert to their numeric-only entry status.

Following is a list of these keys and the functions they perform:

KEY	SAME AS	FUNCTION
1 (End)	Esc--⟩	Bottom of Column
2 ▼	Ctrl--n	Next Line
3 (Pg Dn)	Ctrl--v	Page Down
4 ◀	Ctrl--b	Backward Column
5 No Function		
6 ▶	Ctrl--f	Forward Column
7 (Home)	Esc--⟨	Top of Column
8 ▲	Ctrl--p	Previous Line
9 (Pg Up)	Ctrl--z	Page Up
0 (Ins)	Ctrl--o	Open Line
. (Del)	Ctrl--k	Delete Line

Also of interest:

Esc--0 (Ins)	Esc--o	Open Column
Esc--. (Del)	Esc--k	Delete Column

CHAPTER 8
GETTING STARTED WITH BASIC

Now that you have experimented with the operating system, MS-DOS and have had a chance to work with Perfect Writer and Perfect Calc, which are programmed software packages, it is time to realize the full potential of owning a computer, by creating your own programs. The programming language provided with the Chameleon is Microsoft BASIC.

There are many versions of BASIC, all of them work similarly, but exactly what they can do varies. This chapter will get you started with simple exercises that can be performed with any version of BASIC. To learn the full range of the BASIC supplied with the Chameleon, consult the Microsoft BASIC Reference Manual.

To get into BASIC, boot your working copy of MS-DOS in Drive A. Set the date and time, and wait for the A: to appear. Next key in:

basic86

and hit the Enter key.

Disk drive A will light and whir, and a message will be displayed on your screen indicating the version of Microsoft BASIC you are using. The last line displayed will be:

OK

OK is the BASIC prompt. When you see this you know that BASIC is loaded in the Chameleon and is ready for you to issue a BASIC command or write a program instruction.

BASIC operates in two modes--direct and indirect. In direct mode, you key in a BASIC command, hit the Enter key, and the Chameleon will attempt to execute the command. Try that now by keying:

Print 1+1 ⟨CR⟩

NOTE: ***⟨CR⟩ means press the Carriage Return key after typing the command.***

If all goes well, the Chameleon will respond with:

2
OK

When you see the 2, you will guess that you entered a command that caused the Chameleon to calculate *1+1*, which it did and displayed the answer for you. The OK displayed after the 2 indicates that the computer has executed the instruction and is waiting for you to enter another.

If all did not go well, BASIC will probably respond with:

```
Syntax error
OK
```

If you get this response, it indicates that the computer could not recognize the command, which probably means you misspelled the command.

The command you issued was the ***print*** command. The 1+1 you keyed is an ***operand*** of the print command. Try again. Look at what you key before you press the Enter key. If anything seems to be amiss, use the backspace key (the key with the left facing arrow located above the Enter key) to remove the incorrect characters and re-enter the correct ones. When you think the command and operands are keyed correctly, hit the Enter key again.

It is also possible that the following response could have been given when you attempted to enter the previous example:

```
0
OK
```

This is the response you will see when the command is entered correctly, but BASIC cannot interpret the operand. The print command expects the operands to be: ***a number, a numeric equation that can be solved and/or a character string enclosed in quotation marks.*** In the preceding example, the operand was an equation.

Elaborate on the previous example by keying:

```
print "1 + 1 = " 1+1 (CR)
```

and the computer should respond with:

```
1 + 1 = 2
```

Because you placed the *"1 + 1 = "* in quotation marks, BASIC treated it as a character string and printed it as you keyed it. You could also have put letters and special characters inside the quotes and the machine would display whatever was enclosed by the quotation marks.

In and Out of DOS

Once you are *in BASIC* and the system returns the *OK* prompt, you can no longer use the MS-DOS commands, even though the operating system is still in use. For this reason, some of the often used MS-DOS commands have a counterpart in BASIC. Key the following:

files ⟨CR⟩

You will see that the information returned is similar to the information you would see if you entered *dir* using MS-DOS. The BASIC command *kill* can be used to delete files similarly to the MS-DOS command *erase*.

Once you are familiar with both MS-DOS and BASIC commands you will recognize the similarities between various commands. However, there are not BASIC counterparts for all MS-DOS commands, so if you need to use MS-DOS and you are in BASIC simply type:

system ⟨CR⟩

Try that now and you will see the familiar A: prompt and know that the Chameleon is again under the control of MS-DOS.

To get back into BASIC, key:

basic86 ⟨CR⟩

and you will see the greeting message redisplayed and the *OK* prompt.

So far all of this has been simple, and not very different from the exercises shown in *Chapter 5--Getting Started With MS-DOS*. The real power of BASIC is when it is used to create programs.

PROGRAMMING - The Indirect Mode

In order to create a program, you use BASIC in the *indirect mode*. To signal to BASIC that you are creating a program, simply put a line number before the BASIC command. Then when you hit the Enter key instead of executing the command, as was done in previous exercises, the instruction will be stored in the RAM memory until you either remove it or execute (run) the program. Each of these lines is called a ***program statement.***

BASIC programs contain a number of statements executed in a particular sequence. A statement is similar to a command in that it tells the BASIC program to perform some function. Many statements are created using

commands that can also be issued in the direct mode. In this case the meaning of the statement is the same as the meaning of the command used to create the program statement. There are some statements that cannot be issued as a BASIC command. These are used to control the sequence in which the statements will be performed.

Enough theory, it is time to do some programming.

One of the most important things to remember when programming in BASIC is to begin each programming session by entering the ***new*** command. This clears the RAM memory where your program will be stored. Many unpredictable results can occur, because a programmer entered several lines of code, thinking that a new program was being created, only to discover that the commands were being accepted as part of a program in memory from the last programming session on the computer. So begin by entering:

new ⟨CR⟩

and key the following statement after you get the **OK** prompt:

10 print "1 + 1 = "

At the end of the line review what you keyed to make sure it is what you wanted and press the Enter key. Notice that BASIC did not respond with the prompt ***OK***. The line you key-entered has been stored and BASIC is expecting another line. This time enter:

20 print 1+1 ⟨CR⟩

You will notice that this is similar to the previous exercise and may be wondering how you get the familiar number **2** to be displayed. Entering any BASIC command without the line number will put you back in direct mode so you can do something other than continue your program. Try that now by entering the command:

run ⟨CR⟩

This will direct BASIC to execute whatever program is in memory at the time, and if all has gone according to plan you will see:

```
1 + 1 =
2
OK
```

Another command that is often used to break a programming session is the *list* command. Try that now, enter:

list ⟨CR⟩

What this does is list any program currently in memory.

Now to continue programming simply enter a line number. If you wish to replace a line, enter the number of the line you wish to replace. If you want to add to your program simply use the next number. If you need to insert a line use a number between the two you have already used. Try that now. Add line 30 by entering:

30 print "that is correct" ⟨CR⟩

Notice the *OK* prompt was not returned and you are back in indirect mode.

AUTOmatic Line Numbering

One thing you will find is that on longer programs, keying the line numbers is an annoyance. And on shorter programs, it is easy to forget to key the line numbers and you will find yourself unexpectedly back in direct mode with the last instruction executed instead of stored. Not to worry--BASIC will generate the line numbers for you. To start this process enter the command:

auto ⟨CR⟩

Since you already have a program in memory, BASIC will respond with:

10*

If you hit the Enter key, line 10 will be retained as you keyed it originally and BASIC will respond with:

20*

If you want to change this line simply key the new line. Try keying:

print 1+5 ⟨CR⟩

Press the Enter key again to retain line 30 and finally you will see:

40

This indicates that line 40 did not exist before and BASIC is expecting you to supply it. Try entering:

print "that is not correct, but this is fun" ⟨CR⟩

You have added a line to your program. Now how do you stop BASIC from generating line numbers so you can quit programming and get back into the direct mode. Simply keying a command will not work, because that is what BASIC expects as the content of line 50. To get back into direct mode--so you can instruct the system to run, list, save or anything other than continue programming--you press the Ctrl key and the letter **c** key at the same time. Try that now and you should see the *OK* prompt displayed. This indicates that BASIC is ready to accept a command. Run and list your program, just to see how your program has changed. To run your program Enter:

run ⟨CR⟩

You should now see:

```
1 + 1 =
6
that is correct
that is not correct, but this is fun
```

To list your program Enter:

list ⟨CR⟩

You should now see:

```
10 PRINT "1 + 1 = "
20 PRINT 1 + 1
30 PRINT "that is correct"
40 PRINT "that is not correct, but this is fun"
```

DEBUGGING - Your Syntax Is Showing

You will only see the previous display if you followed the example exactly. If the *operands* (statements) you entered were different from the ones in the example, the lines displayed will reflect those differences. If you misspelled "print" in any of the lines, BASIC will stop executing the program at the line where the misspelling occurred. If the error was in line 30, you would see the following message:

Syntax error in 30

If you get this kind of response, press the Enter key, and BASIC will display the line in error. You can then either rekey the entire statement to correct the error, or use the Edit Mode of BASIC to fix just the parts

that are wrong. Refer to page 46 of the Microsoft BASIC Reference Manual under ***2.16 EDIT*** for Edit Mode commands.

This process of running a program and fixing lines until it works as you expected is known as ***debugging***. If you encountered syntax or logic errors, corrected them, and your program runs without a hitch, you have debugged your first program.

At this point, you have created a simple program that uses only one command and that executes the program lines in straight sequential order. Once you become familiar with the full list of BASIC commands and learn about branches, loops, etc. that control logic flow, you will be able to create programs to solve both mundane and complex problems.

When you begin to build useful or fun programs, you will want to save them so they can be used as needed, or called back up and changed as needed. The program you created in the previous examples is in the Chameleon's RAM and will stay there until:

- you remove the program by using the **new** command,
- you cancel BASIC, by using the **system** command, or
- the power is turned off and all RAM memory is cleared.

Saving And Loading Programs

In order to keep a program that you may want to use again, you can use the **save** command. With this command, you create a data set containing the program, that can be kept on disk. Then when you want to bring the program back into RAM memory, you can use the **new** command to clear memory and the **load** command to transfer the program from the disk to active storage.

You can also load and run a program from a disk when you invoke BASIC by entering the name of the data set at the same time you enter BASIC.

To get some hands on practice with these concepts, you need to save and load a program. Make sure you have your working copy of MS-DOS and BASIC in drive A, ***not*** your original. If BASIC is still invoked and your program is still in RAM from the previous exercises, that is fine. If not, boot your working disk and enter:

basic86 ⟨CR⟩

In either case, enter:

new ⟨CR⟩

to clear the program storage area.

If you wish to use automatic numbering enter:

auto ⟨CR⟩

Now copy the following program into your system:

```
10 print "Greetings"
20 print "Please type in your name and press the Enter key"
30 input a$
40 print a$ ", are you ready to program? If so, enter y."
50 input b$
60 if b$ = "y" then goto 90
70 print "enjoy basic, " a$
80 goto 110
90 print "Enter--auto--and enjoy programming, " a$
100 new
110 end
```

Once you have copied the program, enter **Ctrl** and **c** to exit auto, and get the *OK* prompt. Now you are ready to save your program or debug it. The advantage to saving the program before debugging is that if you lose working lines in trying to correct mistakes, you can go back to your original work by using the **new** and **load** commands to clear memory and reload the program. Save your program now by entering:

save "greeting" ⟨CR⟩

This will save your program under the name **"greeting.bas"**. You can check this using the **FILES** command.

bas is a system supplied extension that indicates that the dataset created contains a BASIC program. If you are using BASIC to program, the ***bas*** will be assumed by BASIC and does not need to be specified in commands. But under MS-DOS or other file handling programs, the full name including the extension must be used to locate the file.

The save command writes the program to the disk, but does not clear RAM, so your program is still in memory. To check this, display the program on the CRT by entering:

list ⟨CR⟩

For the purposes of demonstration, Enter:

new ⟨CR⟩

Now load the program back into memory by Entering:

load "greeting.bas" ⟨CR⟩

List the program as you did before and you should see the same code.

You will notice that six of the lines contain commands that you experimented with in the initial exercises. There are four statements that may be unfamiliar to you.

ENDing It All

The ***end*** statement in line 110 is the simplest statement in the example--it merely marks the end of the program. You do not always need an end statement, because the program will assume the last line in a program is the ***end***.

The reason an ***end*** command was used in this example, was to identify the end of the program to the ***goto*** statement in line 80.

BRANCHING - Or Where Do We GO TO From Here

The ***goto*** statement is a powerful statement used to control the sequence in which program statements are executed. The ***goto*** in line 80 causes the program to skip lines 90 and 100 so it will not print the message in line 90 and will not clear memory as directed in line 100. This is a simple example of program ***branching***.

The ***goto*** statement is even more powerful when it is combined with the ***if--then*** statement as it is in line 60. This statement will bypass lines 70 and 80 in one situation or direct the program to execute lines 70, 80 and 110 in another situation.

IF--THEN - The Fork In the Road

The ***if--then*** statement in line 60 is testing whether or not a variable is equal or a constant. The constant is ***y***. The quotation marks signal to BASIC that ***y*** is a constant. Until the program is changed, line 60 will test some value against the letter y and determine if it is equal. If it is equal

it will follow the instruction to *goto* line 90. If it is not it will perform the next line which, in our example, is line 70.

The variable in line 60 is whatever value is placed in storage at a place known to the program as *b$*. The b indicates an address name, so the program can find the information. The $ indicates that the information is a character, and not a number.

THE INPUT STATEMENT - Online Interaction

By now you should be guessing how the variable information got into the RAM memory addressed by b$. If you decided that the program statement in line 50 has something to do with it, you are right. Line 50 contains the **input** command. This statement will direct the computer to display a question mark and wait for a response. Any information entered in reply will be stored in a location labeled b$. Line 30 also requested information, and saved the information provided in a place labeled a$.

Now that you are familiar with the statements in your program, you are ready to debug it. Start the debugging process by entering:

run ⟨CR⟩

The response you should expect is:

Greetings
Please type in your name and press the Enter key
?

The question mark indicates the input command in line 30 is waiting for your response. Enter:

a-name ⟨CR⟩

The program should respond with:

a-name, are you ready to program?
If so, enter y
?

Depress the Enter key alone, or enter any character other than "y" and the program should respond with:

Enjoy basic, a-name
OK

If all responses match so far proceed, otherwise list your program by entering:

list ⟨CR⟩

and look for differences. Remember you correct lines by re-entering the line you need to replace.

Once you have debugged the negative response to the program, run the program again, and when it displays:

a-name, are you ready to program?
If so, enter y
?

This time enter "y".

The program should now respond with:

Enter--auto--and enjoy programming, a-name
OK

If this is not what you see, you have a challenging debugging problem, because the **new** command in line 100 will clear the program from memory. In order to debug the program you will have to **load** it into memory to **list** it, make your changes, **save** it and **run** it again.

Once you get this program debugged, **save** it one more time. **Note:** each time you save the program with the same name, the original program file is replaced. Each time you save it under a new name, you create a new version. When you are working on a complex program, you will probably want to save programs under a new name until you are satisfied that the most recent version of the program performs better than older versions of the program.

Run Disk, Run

Now that you have saved at least one working version of this sample program, exit BASIC by entering:

system ⟨CR⟩

You should see the A: prompt from MS-DOS. Now, enter:

basic86 greeting ⟨CR⟩

By using ***greeting*** as an operand of the BASIC command, you have directed MS-DOS to load BASIC, and you have directed BASIC to load and run your program. By now you know that you should see:

```
Greetings
Please type in your name and press the Enter key
?
```

Since this little program is not particularly useful, you will probably want to delete it. This can be done in BASIC by entering:

```
kill "greeting.bas"
```

It can also be done by exiting to MS-DOS with the **system** command and using the **erase** command.

Use one of these two methods now and check the deletion using **files** or **dir**.

Although this sample program was small and didn't require a great deal of thought to see how it ran, you can imagine how many statements would be required to do a really useful task through programming.

Learning programming in BASIC is fairly simple. Learning to make the statements perform important functions, in a logical order, quickly, with proper remarks and documentation is an art.

By using the MS-BASIC manual that came with your Chameleon, a lot of practice, and a little patience, you can be turning out programs that will make any task with math, word-processing, accounting, statistics, analysis, or anything you can dream up possible.

Again, congratulations on taking a larger view of the BIG picture by purchasing your brand new Chameleon.

APPENDIX - A
ASCII CODE CHART

CHAR	DEC	HEX	CHAR	DEC	HEX
NUL	000	00H	SOH	001	01H
STX	002	02H	ETX	003	03H
EOT	004	04H	ENQ	005	05H
ACK	006	06H	BEL	007	07H
BS	008	08H	HT	009	09H
LF	010	0AH	VT	011	0BH
FF	012	0CH	CT	013	0DH
SO	014	0EH	SI	015	0FH
DLE	016	10H	DC1	017	11H
DC2	018	12H	DC3	019	13H
DC4	020	14H	NAK	021	15H
SYN	022	16H	ETB	023	17H
CAN	024	18H	EM	025	19H
SUB	026	1AH	ESCAPE	027	1BH
FS	028	1CH	GS	029	1DH
RS	030	1EH	US	031	1FH
SPACE	032	20H	!	033	21H
"	034	22H	#	035	23H
$	036	24H	%	037	25H
&	038	26H	'	039	27H
(	040	28H	)	041	29H
*	042	2AH	+	043	2BH
,	044	2CH	−	045	2DH
.	046	2EH	/	047	2FH
0	048	30H	1	049	31H
2	050	32H	3	051	33H
4	052	34H	5	053	35H
6	054	36H	7	055	37H
8	056	38H	9	057	39H
:	058	3AH	;	059	3BH
<	060	3CH	=	061	3DH
>	062	3EH	?	063	3FH

CHAR	DEC	HEX	CHAR	DEC	HEX
@	064	40H	A	065	41H
B	066	42H	C	067	43H
D	068	44H	E	069	45H
F	070	46H	G	071	47H
H	072	48H	I	073	49H
J	074	4AH	K	075	4BH
L	076	4CH	M	077	4DH
N	078	4EH	O	079	4FH
P	080	50H	Q	081	51H
R	082	52H	S	083	53H
T	084	54H	U	085	55H
V	086	56H	W	087	57H
X	088	58H	Y	089	59H
Z	090	5AH	[	091	5BH
\	092	5CH	]	093	5DH
^	094	5EH	_	095	5FH
`	096	60H	a	097	61H
b	098	62H	c	099	63H
d	100	64H	e	101	65H
f	102	66H	g	103	67H
h	104	68H	i	105	69H
j	106	6AH	k	107	6BH
l	108	6CH	m	109	6DH
n	110	6EH	o	111	6FH
p	112	70H	q	113	71H
r	114	72H	s	115	73H
t	116	74H	u	117	75H
v	118	76H	w	119	77H
x	120	78H	y	121	79H
z	122	7AH	{	123	7BH
|	124	7CH	}	125	7DH
~	126	7EH	DEL	127	7FH

APPENDIX - B
ASSEMBLY LANGUAGE INSTRUCTION SET

8088

The following is an alphabetical listing of the 8088 instruction set mnemonics, their relative machine codes, and full name descriptions.

MNEM	MACHINE	FUNCTION
AAA	0011 0111	ASCII adjust for addition
AAD	1101 0101	ASCII adjust for division
AAM	1101 0100	ASCII adjust for multiplication
AAS	0011 1111	ASCII adjust for subtraction
ADC	0001 000w	Add with carry
ADD	0000 000w	Add
AND	0010 000w	And
CALL	1110 1000	Call
CBW	1001 1000	Convert byte to word
CLC	1111 1000	Clea carry flag
CLD	1111 1100	Clear direction flag
CLI	1111 1010	Clear interrupt flag
CMC	1111 0101	Compare
CMP	0011 10dw	Compare byte or word
CMPS	1010 011w	Compare byte string
CWD	1001 1001	Convert word to double word
DAA	0010 0111	Decimal adjust for addition
DAS	0010 1111	Decimal adjust for subtraction
DEC	1111 111w	Decrement
DIV	1111 011w	Divide
ESC	1101 1000	Escape
HLT	1111 0100	Halt
IDIV	1111 011w	Integer divide
IMUL	1111 011w	Integer multiply
IN	1110 010w	Input byte or word
INC	0100 0000	Increment
INT	1100 1101	Interrupt
INTO	1100 1110	Interrupt on overflow
IRET	1100 1111	Interrupt return
JA	0111 0111	Jump on above
JAE	0111 0011	Jump on above or equal
JB	0111 0010	Jump on below
JBE	0111 0110	Jump on below or equal

MNEM	MACHINE	FUNCTION
JCXZ	1110 0011	Jump on CX zero
JE	0111 0100	Jump on equal
JG	0111 1111	Jump on greater
JGE	0111 1101	Jump on greater or equal
JL	0111 1100	Jump on less than
JLE	0111 1110	Jump on less than or equal
JMP	1110 1001	Jump
JNA	0111 0110	Jump on not above
JNAE	0111 0010	Jump on not above or equal
JNB	0111 0011	Jump on not below
JNBE	0111 0111	Jump on not below or equal
JNE	0111 0101	Jump on not equal
JNG	0111 1110	Jump on not greater
JNGE	0111 1100	Jump on not greater or equal
JNL	0111 1101	Jump on not less than
JNLE	0111 1111	Jump on not less than or equal
JNO	0111 0001	Jump on not overflow
JNP	0111 1011	Jump on not parity
JNS	0111 1001	Jump on not sign
JNZ	0111 0101	Jump on not zero
JO	0111 0000	Jump on overflow
JP	0111 1010	Jump on parity
JPE	0111 1010	Jump on parity even
JPO	0111 1011	Jump on parity odd
JS	0111 1000	Jump on sign
JZ	0111 0100	Jump on zero
LAHF	1001 1111	Load AH with flags
LDS	1100 0101	Load pointer into DS
LEA	1000 1101	Load effective address
LES	1100 0100	Load pointer into ES
LOCK	1111 0000	Lock bus
LODS	1010 110w	Load byte or word
LOOP	1110 0010	Loop
LOOPE	1110 0001	Loop while equal
LOOPNE	1110 0000	Loop while not equal
LOOPNZ	1110 0000	Loop while not zero
LOOPZ	1110 0001	Loop while zero
MOV	1000 10dw	Move

MNEM	MACHINE	FUNCTION
MOVS	1010 010w	Move byte or word
MUL	1111 011w	Multiply
NEG	1111 011w	Negate
NOP	1001 0000	No operation
NOT	1111 011w	Not
OR	0000 100w	Or
OUT	1110 011w	Output byte or word
POP	1000 1111	Pop
POPF	1001 1101	Pop flags
PUSH	1111 1111	Push
PUSHF	1001 1100	Push flags
RCL	1101 00vw	Rotate through carry left
RCR	1101 00vw	Rotate through carry right
REP	1111 0010	Repeat
RET	1100 0011	Return
ROL	1101 00vw	Rotate left
ROR	1101 00vw	Rotate right
SAHF	1001 1110	Shift AH into flags
SAL	1101 00vw	Shift arithmetic left
SAR	1101 00vw	Shift arithmetic right
SBB	0001 10dw	Subtract with borrow
SCAS	1010 111w	Scan byte or word
SHL	1101 00vw	Shift left
SHR	1101 00vw	Shift right
STC	1111 1001	Set carry flag
STD	1111 1100	Set direction flag
STI	1111 1011	Set interrupt flag
STOS	1010 101w	Store byte or word
SUB	0010 10dw	Subtract
TEST	1000 010w	Test
WAIT	1001 1011	Wait
XCHG	1000 011w	Exchange
XLAT	1101 0111	Translate
XOR	0011 00dw	Exclusive Or

w - word operation
v - variable
d - direct

Z80A

The following is an alphabetical listing of the Z8OA instruction set mnemonics and full name descriptions.

MNEMONIC	FUNCTION
ADC A	Add with carry to accumulator
ADC HL,rp	Add register pair with carry to HL
ADD	Add
AND	And, logical
BIT	Test bit
CALL addr	Call, subroutine
CALL cond.addr	Call, conditional
CCF	Complement carry flag
CP	Compare
CPD	Compare, decrement
CPDR	Compare, decrement-repeat
CPI	Compare, increment
CPIR	Compare, increment-repeat
CPL	Complement accumulator
DAA	Decimal adjust accumulator
DEC	Decrement
DJNZ	Decrement and jump if not zero
DI	Disable interrupts
EI	Enable interrupts
EX	Exchange
EXX	Exchange register pairs & alternatives
HALT	Halt
IM	Set interrupt mode
IN	Input
INC	Increment
IND	Input, decrement
INDR	Input, decrement-repeat
INI	Input, increment
INIR	Input, increment-repeat
JR	Jump, relative
JR cond.addr	Jump, relative conditional
JP addr	Jump
JP cond.addr	Jump, conditional

MNEMONIC	FUNCTION
LD reg/(HL)	Load register
LD A,(addr)	Load accumulator direct
LD data	Load, immediate
LD A,(BC) or (DE)	Load accumulator secondary
LD HL,(addr)	Load HL direct
LD reg,(xy+disp)	Load register indexed
LD rp,(addr)	Load register pair direct
LD xy,(addr)	Load index register direct
LDD	Load, decrement
LDDR	Load, decrement-repeat
LDI	Load, increment
LDIR	Load, increment-repeat
LD A,I	Load accum. from interrupt vector reg.
LD A,R	Load accum. from refresh register
LD (HL),reg	Store register
LD (addr),A	Store accumulator direct
LD (BC) or (DE),A	Store accumulator secondary
LD (addr),HL	Store HL direct
LD (xy+disp),reg	Store register indexed
LD (addr),rp	Store register pair direct
LD (addr),xy	Store index register direct
LD (HL),data	Store immediate to memory
LD (xy+disp),data	Store immediate to memory indexed
LD I,A	Store accum. to interrupt vector reg.
LD R,A	Store accum. to refresh register
LD SP,HL	Move HL to stack pointer
LD SP,xy	Move index register to stack pointer
LD dst,src	Move, register to register
NEG	Negate accumulator,(2's complement)
NOP	No operation
OUT	Output
OR	Or, logical
OUTD	Output, decrement
OTDR	Output, decrement-repeat
OUTI	Output, increment
OTIR	Output, increment-repeat
POP	Pop from stack
PUSH	Push to stack

MNEMONIC	FUNCTION
RES	Reset bit
RET	Return from subroutine
RET cond	Return, conditional
RETI	Return from interrupt
RETN	Return from non-maskable interrupt
RL	Rotate left through carry
RLA	Rotate accumulator-left through carry
RLC	Rotate left circular
RLCA	Rotate accumulator left circular
RLD	Rotate accum. & memory left decimal
RR	Rotate right through carry
RRA	Rotate accumulator-right through carry
RRC	Rotate right circular
RRCA	Rotate accumulator right circular
RRD	Rotate accum. & memory right decimal
RST	Restart
SBC	Subtract with carry (borrow)
SCF	Set carry flag
SET	Set bit
SLA	Shift left arithmetic
SRA	Shift right arithmetic
SRL	Shift right, logical
SUB	Subtract
XOR	Exclusive or, logical

APPENDIX - C
I/O PORT LOCATIONS

PORT	FUNCTION
0000 - 00FH	8237 DMA CONTROLLER
010H - 017H	8291 IEEE-488 TALKER/LISTENER
020H - 02FH	8259 INTERRUPT CONTROLLER
030H - 031H	8274 SERIAL PORT CONTROLLER
040H - 043H	8253 INTERVAL TIMER
050H - 051H	8292 IEEE-488 CONTROLLER
060H - 06FH	KEYBOARD AND CONTROL PORT
080H	DMA PAGE REGISTER - WRITE ONLY
201H	GAME STATUS PORT - READ
	START GAME TIMERS - WRITE
37CH	PARALLEL PRINTER DATA
37DH	PARALLEL PRINTER STATUS - READ
	Z-80 MEMORY BANK REGISTER - WRITE
37EH	PARALLEL PRINTER STATUS - READ
	CONTROL - WRITE
3D4H - 3D5H	6845 CRT CONTROLLER
3D8H	CRT MODE SELECT REGISTER - WRITE ONLY
3D9H	CRT COLOR SELECT REGISTER - WRITE ONLY
3DAH	CRT STATUS - READ ONLY
3DBH	LIGHT PEN,
	ANY READ/WRITE CLEARS FLIP-FLOP
3DCH	LIGHT PEN,
	ANY READ/WRITE SETS FLIP-FLOP
3DDH	OPTION REGISTER 1, WRITE-ONLY, 8 BITS
3DEH	PROCESSOR SWAP STROBE ADDR
3DFH	OPTION REGISTER 3, WRITE-ONLY, 4 BITS
3F2H	FLOPPY-DISK CONTROL REGISTER, WRITE-ONLY
3F4H	765 DISK CONTROLLER STATUS REGISTER
3F5H	765 DISK CONTROLLER DATA REGISTER

For registers that state "WRITE-ONLY", a READ operation will destroy the data in the register.

NOTE: The Chameleon decodes 11 address lines for I/O ports, allowing up to 2,048 I/O devices total. Of these, 176 are used on the Chameleon, leaving 1,872 available.

APPENDIX - D
MEMORY MAP

ON-BOARD MEMORY USE

STARTING ADDRESS HEX

0-3FFFFH |

256 K BYTES DYNAMIC RAM

40000H-AEFFFH |

444 K EXTERNAL EXPANSION RAM
(Optional)

AF000H-AFFFFH |

4 K BYTE CHARACTER-GENERATOR RAM

B8000H-BBFFFH |

16 K BYTE DISPLAY MEMORY, with 16 K-bit RAMs

F3000H-F303FH |

64 NIBBLES (D0-D3), NONVOLATILE RAM (EEROM)
(Repeats 8 times from F3000H-F31FFH)

F3200H |

A READ operation transfers the non-volatile ROM to its RAM area. A WRITE operation transfers the RAM area to the ROM area for non-volatile recall.
(Repeats from F3200H-F33FFH)

F3400H-F3403H |

4 NIBBLES (D0-D3), GRAPHICS COLOR LOOK-UP TABLE 0
(Repeats 128 times from F3400H-F35FFH)

F3600H-F3603H |

4 NIBBLES (D0-D3), GRAPHICS COLOR LOOK-UP TABLE 1
(Repeats 128 times from F3600H-F37FFH)

(F4000H-FFFFFH) |

- 48 K BYTES OF ROM, with 8 K byte ROM option selected. ROM does not repeat in its memory space.
- 24 K BYTES OF ROM, with 4 K byte ROM option selected. ROM starting at FA000H

Memory blocks which are not listed in the table are available to external memory devices.

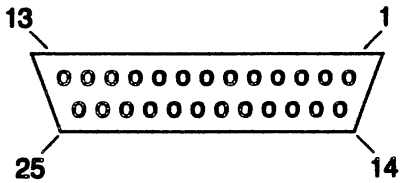
NOTE: REPEATS means that the memory block will appear more than once within its memory space. It can be accessed at any of its repetitive locations.

APPENDIX - E
I/O PORT PINOUTS

The cables that are utilized on the Chameleon parallel port are identical to IBM PC parallel port cables (same pinout configuration). The serial port has some minor differences. The following are diagrams of the actual wiring of the cables.

PARALLEL PORT

The following diagram illustrates the output signal of the Chameleon when connected to a Centronix-compatible printer.

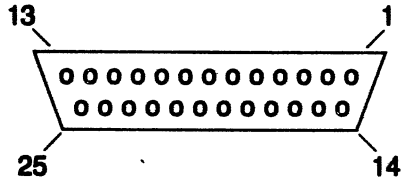


Female (Port out of Chameleon)

Pin #	Signal Name	Amphenol 36-pin (male)
1	−Strobe	1
2	+Data Bit 0	2
3	+Data Bit 1	3
4	+Data Bit 2	4
5	+Data Bit 3	5
6	+Data Bit 4	6
7	+Data Bit 5	7
8	+Data Bit 6	8
9	+Data Bit 7	9
10	−Acknowledge	10
11	+Busy	11
12	+Out of Paper	12
13	+Select	13
14	−Auto Feed	14
15	−Error	32
16	−Initialize Printer	31
17	−Select Input	36
18	Ground	19,21,23,25,27,29,30,33

SERIAL PORT

The following diagram illustrates the output signal of the Chameleon when connected to an Asynchronous Communications Adapter (RS232C) or serial printer.



Female (Port out of Chameleon)

Chameleon	Port Pin # (w/signal direction)	External Device (Signal Name)
GND	1	Frame Ground
RDX	2 ←	Transmit Data
TXD	3 →	Receive Data
DCD	4 ←	Request to Send
DTR	5 →	Clear to Send
RTS	6 →	Data Set Ready
GND	7	Signal Ground
RTS	8 →	Carrier Detect
CTS	20 ←	Data Terminal Ready

APPENDIX - F
SAMPLE CABLES

The following cable conversion table is provided for various pin configurations of different printer manufacturers.

CHAM = CHAMELEON OUTPUT (female D-SHELL on rear of computer).

CENT = Centronics - Amphenol type 47,
Part# 47-30360 connector.

AX = AXIOM EX800 - DB-25 Male connector,
ITT-CANNON, Part# DB25P,
RS232 interface.

DW = DAISY WRITER - DB-25P

MNEMONIC	CHAM	CENT	AX	DW
-STROBE	1	1	24	1
+DP0	2	2	15	2
+DP1	3	3	16	3
+DP2	4	4	17	4
+DP3	5	5	18	5
+DP4	6	6	19	6
+DP5	7	7	21	7
+DP6	8	8	23	8
+DP7	9	9	--	9
-ACK	10	10	14	10
BUSY	11	11	--	11
PAPER ERROR	12	12	--	-
SELECT	13	13	--	13
AUTO LF	14	14	--	--
-ERROR	15	32	--	--
-INIT	16	31	--	--
-SELECT IN	17	36	--	--
GROUND	18	19	7	19
"	19	21	--	17*
"	20	23	--	-
"	21	25	--	-
"	22	27	--	-
"	23	29	--	-
"	24	30	--	-
"	25	33	--	-

* - Daisy Writer pin 17 is chassis ground. Pins 18-25 are jumpered on the DB-25P connector with bus wire.

The following information is provided as a reference only. Manufacturers have the option of changing the method by which their equipment operates, so all data may not be compatible with newer or revised models.

CENTRONICS COMPATIBLE PRINTERS:

Centronics

Okidata - 82, 83, 92, 93, 84

Epson - MX80, MX80FT, FX80, RX80, MX100

Transtar - 120, 130, 140 (letter quality)

Diablo - 620, 630 parallel (letter quality)

Riteman

Gemini

Prowriter

Starwriter (letter quality)

NEC 3550 (letter quality)

DAISYWRITER:

KELAM Part# RC250142-5.

Black dot on Daisy Writer connector indicates Pin # 1. All odd pins are on one side and even numbered pins on the opposite side.

PIN#	FUNCTION	SIGNAL DIRECTION
9	SELECT	OUT
11	BUSY	OUT
12	GROUND	IN/OUT
13	ACK	OUT
17	DATA0	IN
19	DATA1	IN
21	DATA2	IN
23	DATA3	IN
25	DATA4	IN
27	DATA5	IN
29	DATA6	IN
31	DATA7	IN
40	STROBE RTN	IN/OUT
41	STROBE	IN
46	FRAME GND	IN/OUT

APPENDIX - G
CHAMELEON ROM BIOS MONITOR

The Chameleon ROM BIOS has a built-in monitor. This can be used for machine level functions. The monitor is automatically activated when the Chameleon is RESET or BOOTED without a disktette in drive A.

COMMANDS:

B = Boot diskette in Drive A
C = Reset Special Parameters
D(seg:addr) = Dump 8 bytes starting at the Segment and Address specified
E(seg:addr data1) = Enter hex data starting at Segment and Address specified
G(seg:addr) = Execute starting at Segment and Address specified
I(port) = Input 1 byte from port specified
L(seg:addr dr sc tk hd) = Load Sector specified at Segment and Address specified
O(port data1) = Output 1 byte to port specified
V(mode) = Select new video mode (0 thru 6)
W(seg:addr dr sc tk hd) = Write data at Segment and Address specified to Sector specified

seg = Segment in Hex (0H to FFFFH)
addr = Address in Hex (0H to FFFFH)
data1 = Data Byte Hex
port = I/O port (0H to 7FFH Hex)
mode = Video Mode (0 to 6)
dr = Drive Number (0 or 1)
sc = Sector Number (1 to 8)
tk = Track Number (0 to 27 Hex)
hd = Head Number (0 or 1)

* Note: No space should be left between the command and the first option.

ROM BIOS ENTRY POINTS

The ROM Basic I/O System (BIOS) is a set of subroutines to provide easy programming access to the input and output devices in the Chameleon. These are things such as the keyboard, video, disk drives, etc. These subroutines are accessed through the use of interrupt vectors and the passing of parameters.

The following listing contains the majority of usable interrupt vectors for the Chameleon ROM BIOS.

INTERRUPT VECTOR LISTING

Interrupt #	Interrupt Name
0	Divide by Zero
1	Single Step
2	Non Maskable
3	Breakpoint
4	Overflow
5	Print Screen
6	Unused
7	Unused

8259 Interrupt Vectors

Interrupt #	Interrupt Name
8	Time of Day
9	Keyboard
A	Unused
B	Unused
C	Reserved for Communications
D	Unused
E	Diskettes I/O
F	Reserved for Printer

BIOS Entry Points

Interrupt #	Interrupt Name
10	Video
11	Equipment Check
12	Memory Size
13	Disk I/O
14	Communications
15	Reserved
16	Keyboard I/O
17	Printer I/O

Interrupt #	Interrupt Name
18	Reserved
19	Bootstrap
1A	Time of Day
1B	Keyboard Break
1C	Timer Tick
1D	Video Initialization
1E	Disk Parameters
1F	Graphics Characters

∇ Interrupt 05H PRINT SCREEN

Invoking this interrupt will cause the current text screen to be copied to the current list device. The status of the operation is stored at memory location 50:0 hex.

Status:

0 = Screen print successfully completed
1 = Screen print in progress
255 = Error during screen print

∇ Interrupt 10H VIDEO I/O

These routines provide the interface for the CRT in both Text and Graphics modes.

- AH = 00H Set Video Mode
  - AL = 0 Text 40 x 25 B&W
  - AL = 1 Text 40 x 25 Color
  - AL = 2 Text 80 x 25 B&W
  - AL = 3 Text 80 x 25 Color
  - AL = 4 Graphics 320 x 200 Color
  - AL = 5 Graphics 320 x 200 B&W
  - AL = 6 Graphics 640 x 200 B&W
- AH = 01H Set Cursor Size
  - CH = Bits 0 to 4 Cursor Left Side
  - CL = Bits 0 to 4 Cursor Right Side

* Note: Setting Bits 5, 6 and 7 may cause erratic or no cursor

- AH = 02H Set Cursor Position (0,0 = Upper Left)
  - DH = Cursor Row
  - DL = Cursor Column
  - BH = Page Number (Must be 0 for Graphics Modes)
- AH = 03H Read Cursor Position
  - BH = Page Number

  At Exit:
  - DH = Cursor Row
  - DL = Cursor Column
  - CX = Cursor Size
- AH = 04H Read Light Pen Location

  At Exit:

  If AH = 0 Light Pen not active
  If AH = 1 Light Pen Position Read

  Text Mode:
  - DH = Light Pen Row
  - DL = Light Pen Column

  Graphics Mode:
  - CH = Light Pen Row
  - BX = Light Pen Column
- AH = 05H Set Video Page (Text Modes Only)
  - AL = New Page

  Valid Pages:

  0 to 7 for modes 0 & 1
  0 to 3 for modes 2 & 3
- AH = 06H Scroll Current Video Page Up
  - AL = Number of line to scroll (0 = whole window)
  - CH = Row of Upper left corner of window
  - CL = Column of Upper Left corner of window
  - DH = Row of lower right corner of window
  - DL = Column of lower right corner of window
  - BH = Attribute for blank lines

- AH = 07H Scroll Current Video Page Down
  - AL = Number of line to scroll (0 = whole window)
  - CH = Row of Upper left corner of window
  - CL = Column of Upper Left corner of window
  - DH = Row of lower right corner of window
  - DL = Column of lower right corner of window
  - BH = Attribute for blank lines
- AH = 08H Read Character and Attribute at Cursor
  - BH = Video Page (Text Modes Only)

  At Exit:
  - AL = Character
  - AH = Attribute
- AH = 09H Write Character and Attribute at Cursor
  - BH = Video Page (Text Modes Only)
  - CX = Character Count
  - AL = Character
  - BL = Attribute
- AH = 0AH Write Character Only
  - BH = Video Page (Text Modes Only)
  - CX = Character Count
  - AL = Character
- AH = 0BH Set Color Palets

  Set Backgound Color
  - BH = 0
  - BL = Background Color (0 to 15)

  Select Color Palet
  - BH = 1
  - BL = Color Palet (0 or 1)
- AH = 0CH Write Graphic Dot
  - DX = Row Number
  - CX = Column Number
  - AL = Color

  * Note: If Bit 7 of AL = 1 then the new color is Exculsive OR'd with the color of the current dot.

■ AH = 0DH Read Graphic Dot

- ■ DX = Row Number
- ■ CX = Column Number

At Exit:

- ■ AL = Color Value of Dot

■ AH = 0EH Write Teletype Style

- ■ AL = Character to write
- ■ BL = Foreground Color (Graphics Modes Only)
- ■ BH = Video Page (Text Modes Only)

■ AH = 0FH Get Current Video Mode

At Exit:

- ■ AL = Current video mode
- ■ AH = number of screen columns
- ■ BH = Current Vide4o Page

∇ Interrupt 11H GET EQUIPMENT INFO

This routine returns the current configuration of the system.

At Exit:

- ■ AX = Returns current equipment

Bit 0 = Startup Mode
0 = goto ROM monitor at reset
1 = Boot diskette at reset
Bit 1 = Not Used
Bit 2 & 3 = Base Ram Size
(Always 11)
Bit 4 & 5 = Initial Video Mode
01 = 40 x 25 Color
10 = 80 x 25 Color
11 = 80 x 25 B&W
Bit 6 & 7 = Number of Disk Drives
Bit 8 = Unused
Bit 9,10 & 11 = Number of Serial Ports
Bit 12 = Game I/O Attached
Bit 13 = Unused

Bit 14 & 15 = Number of Printers

∇ Interrupt 12H GET RAM SIZE

This routine returns the number of contiguous 1K blocks of RAM.

At Exit:

- AX = Number of 1K blocks

∇ Interrupt 13H DISKETTE I/O

These routines provide the I/O for the disk drive operations.

- AH = 0 Reset the Disk Drives
- AH = 1 Get Status of the Disk System

At Exit:

- AL = Status

- AH = 2 Read Sector(s)
- AH = 3 Write Sector(s)
- AH = 4 Verify Sector(s)
- AH = 5 Format a Track

At Entry: Read, Write, Verify and Format Parameters

- DL = Drive Number (0 to 3)
- DH = Head Number (0 or 1)
- CH = Track Number (0 to 27H)
- CL = Sector Number (1 to 8)
- AL = Number of Sectors (8 Maximum)
- ES:BX = Address of the R/W Buffer

At Exit:

- AH = Status
- AL = Number of Sectors Read
- CY = Good/Bad Operation

∇ Interrupt 14H COMMUNICATIONS I/O (RS-232)

This routine provides the I/O routines for using the Communications port.

- AH = 0 Initialize Port
  - DX = Port Number
  - AL = Parameters

  Word Length

  Bits 1,0 = 10 (7 Bits)
  = 11 (8 Bits)

  Stop Bit

  Bit 2 = 0 (1 Stop Bits)
  = 1 (2 Stop Bits)

  Parity

  Bits 4,3 = 00 (None)
  = 01 (Odd)
  = 11 (Even)

  Baud Rate

  Bit 7,6,5 = 000 (110 Baud)
  = 001 (150 Baud)
  = 010 (300 Baud)
  = 011 (600 Baud)
  = 100 (1200 Baud)
  = 101 (2400 Baud)
  = 110 (4800 Baud)
  = 111 (9600 Baud)

- AH = 1 Transmit Character
  - DX = Port Number
  - AL = Character to be sent

  At Exit:

  - AH = Status of operation

- AH = 2 Receive Character
  - DX = Port Number

  At Exit:

  - AL = Character Sent
  - AH = Status

- AH = 3 Status of the COMM Port
  - DX = Port Number

  At Exit:
  - AH = Line Control Status

    bit 7 = Time Out
    bit 6 = Shift reg. empty
    bit 5 = Holding reg. empty
    bit 4 = Break
    bit 3 = Framing err.
    bit 2 = Parity err.
    bit 1 = Overrun err.
    bit 0 = Data ready
  - AL = Modem Status

    bit 7 = Line detect
    bit 6 = Ring
    bit 5 = Data set ready
    bit 4 = Clear to send
    bit 3 = Delta line detect
    bit 2 = Trailing edge
    bit 1 = Delta Data set ready
    bit 0 = Delta clear to send

∇ Interrupt 16H KEYBOARD I/O

These routines provide access to the keyboard.

- AH = 0 Read the next character from the keyboard

  At Exit:
  - AL = Character
  - AH = Scan Code
- AH = 1 Check if character ready

  At Exit:

  If ZF = 1 no character available
  If ZF = 0 character available
- AH = 2 Get Current Shift Status

  At Exit:
  - AL = Shift code

∇ Interrupt 17H PRINTER I/O

These routines provide support for the parallel printer port.

- AH = 0 Print Character
  - AL = Character to be printed
  - DX = Port to be used

  At Exit:

  If AH = 1 then no character printed
- AH = 1 Initialize Port
  - DX = Port to be used

  At Exit:
  - AH = Printer status
- AH = 2 Get Printer Status
  - DX = Port to be used

  At Exit:
  - AH = Printer Status

    bit 0 = Time out
    bit 2,1 = Not Used
    bit 3 = I/O error
    bit 4 = Select
    bit 5 = Paper out
    bit 6 = Acknowledge
    bit 7 = Busy

∇ Interrupt 19H DISK BOOT STRAP

This routine boots the current diskette in drive A. If there is no diskette in Drive A or it does not contain a valid boot the ROM monitor will be operative.

∇ Interrupt 1AH TIME OF DAY

This routine allows you to read or set the system clock.

- AH = 0 Read the Clock

At Exit:

- CX = High word of the TIME
- DX = Low word of the TIME

If ■ AL = 0 timer has not passed 24 hours

■ AH = 1 Set the Clock

- CX = High word of the TIME
- DX = Low word of the TIME

APPENDIX - H
CTERM Smart Terminal Simulator

GENERAL DESCRIPTION

CTERM is a communication package program provided on the MS-DOS disk and is intended for use on the serial port of the Chameleon computer. Its primary functions are to provide the capability of uploading and downloading text files from one computer to another through a modem device. It is also capable of sending and receiving non-text files. The serial port interrupt driver (SERDRV.OBJ) is utilized by CTERM for communications.

NOTE: At this time there is no good way to detect any transmission errors that might occur. In future versions the package will include XMODEM error detection and correction protocol.

FUNCTION KEY DESCRIPTION

F1 = EXIT

Function key 1 is used to exit from the terminal package to the operating system.

WARNING: All files must be closed before exiting to prevent the loss of any information in the file currently being spooled. See function keys 5 & 6 for more information.

F2 = FLUSH

Function key 2 has two basic functions. One is to blank the screen and clear the buffers of all information. The second is for critical error recovery. Whenever the receiver buffer overflows, a message will prompt the user to please FLUSH the buffers. Under normal conditions, this will never happen. But at 9600 baud, and after approximately 100 K characters have been received, and spooling is taking place, the receiver buffer will overflow losing part of the transmission.

F3 = PARM

Function key 3 will allow you to change the parameters of the transmission protocol currently being used to send and receive information over the serial port. These parameters include setting the baud rate, word length, use of one or two stop bits and whether parity checking is odd, even or not in effect. CTERM defaults to the values set by the OPTION program.

WARNING: Care should be taken not to change the current parameters while transmitting or spooling a file. All buffers are cleared and information which is cued up is lost.

F4 = DUPL

Function key 4 is the duplex mode switch. In the half duplex mode information is echoed to the screen by the user system when it is being placed in the transmission buffer. This includes both when a key is typed and when the file transmit function (F7) is enacted. In the full duplex mode there is no echoing of information done by the user's system to the screen. Either the modem or host system must be set up to accomplish this. The duplex mode should be agreed upon in advance by both the host and the user.

F5 = SPOOL

Function key 5 is used during reception to open a new disk file for spooling an incoming file. Spooling, which is short for simultaneous peripheral operations on line, is used to speed up reception of data from another system by placing it on a disk. This is much quicker than printing to the screen or a printer. When the spooling function is selected, all incoming information is cued up in the receiving buffer while the program prompts the user to enter a file name to store the data in. Once a valid file name has been accepted and one record's worth of information is received, it is written to the disk or any valid MS-DOS device.

NOTE: A spooled file may be lost if it is not properly closed by Function 6 before exiting the communications program to the operating system.

F6 = STOPS

Function key 6 is used to close out the current file being spooled to and allows a new file to be opened.

F7 = XMIT

Function key 7 is used to open a file on disk or other peripheral device recognized by MS-DOS and start transmitting it over the serial port to another system. The user is first prompted to enter a valid file for transmission. The outgoing transmission can be monitored in the half duplex mode. Incoming information can be monitored in the full duplex mode assuming no one is echoing the outgoing transmission back to the user's system.

F8 = STOPX

Function key 8 stops the transmission of the current file when the transmit buffer has been emptied.

BUFFER DESCRIPTION

The Transmit buffer is 4,000 bytes in length, but only 512 bytes of it at a time are transmitted to the receiving system.

The Receiver buffer is 64,000 bytes in length and is fully utilized for normal reception and spooling.

NOTE: Both buffers are circular-type buffers. The serial port driver used by CTERM is SERDRV.OBJ which is included with the operating system. See *APPENDIX - J SERDRV.OBJ* for a detailed description of how it is used.

APPENDIX - I
FILCOM File Compare Utility

GENERAL DESCRIPTION

FILCOM, which is located on the MS-DOS disk, is a text and assembly file comparison utility program. Its purpose is to determine if two files hold identical or dissimilar data. The results of the comparison can be directed to your screen (CON), a parallel printer (LST), a serial printer or device (AUX), or written to a .DIF file for later viewing.

If you will follow the steps below, you should be able to fully comprehend and use the FILCOM utility quickly.

Place the MS-DOS disk which contains the FILCOM dataset in Drive A and boot the disk if you haven't already done so.

With the A: prompt on the screen type:

filcom ⟨CR⟩

Note: ⟨CR⟩ represents pressing the Enter key.

The screen should now show the following:

Microsoft File Compare Utility [21-JAN-82]
Copyright © 1981 by Microsoft, Inc.

Source1 filename [.ASM]:

The program is requesting the name of the first file you wish to compare. The [.ASM] is the default file extension signifying an assembly language file. If the file you wish to compare *is* an assembly file you can simply type the filename without the delimiter (.) and the extension (asm).

For this example, you will compare two files on the second MS-DOS disk. You may either place it in drive B or remove the disk in drive A and replace it with the 2 of 2 disk.

Note: If you use drive B, you must key the drive specifier b: as part of the filename.

Now, type:

serdrv.obj ⟨CR⟩ if you are using drive A

or

b:serdrv.obj ⟨CR⟩ if you are using drive B

and you should see:

Source2 filename [SERDRV.BAK]:

or

Source2 filename [B:SERDRV.BAK]:

The program is now asking for the name of the file you wish to compare the first file to. It shows a default of SERDRV.BAK because it assumes you wish to compare a file with its backup. If this were the case, you could simply type a ⟨CR⟩ if the second file were named exactly as the default appears. If not, then Enter in the correct name. For this example key:

serdrv.doc ⟨CR⟩

You should now see:

List filename [SERDRV.DIF]:

or

List filename [B:SERDRV.DIF]:

The program is asking where you would like the output results to be displayed. The default SERDRV.DIF assumes that the output should be placed in a .DIF file for viewing at a later time. If you wish to view the output immediately, Enter:

con ⟨CR⟩

which will display directly to the console (CRT).

If you wish to view the output in printed form (assuming you have a parallel printer correctly attached), Enter:

lst ⟨CR⟩

If you wish to direct the output to a serial printer or a modem or another computer (assuming a serial device is correctly attached), Enter:

aux ⟨CR⟩

Once you have directed the output to the desired location and pressed the Enter key, the comparison of the two files should begin.

If you happened to use an incorrect input character in any of the preceding entries , such as an Escape key or Ctrl command, you should have received the following:

?Command error: 'x'

and been returned to the input prompt where the error was made. The *x* in single quotes will be one of 128 graphic characters and will be different depending on which commands you input.

If you should see:

?File not found

then one of the files you are comparing was misspelled, the drive specifier was missing or not correct, or the file does not exist. It should also be mentioned that a file without a file extension is not valid to the program. You will be prompted again for the Source1 filename.

The message:

Files are different

tells you that either the two files do not match; or that one or both of the files were not text or assembly files. These two types of files are the only kind that can be compared with this utility.

Barring any of the preceding mishaps, the program checks both files. (You may notice a delay at this point depending on the length of the two files. The bigger the files, the longer the delay.)

Depending on where you directed your output, one of the following will take place (Note: the two files used in this example will be entirely different and therefore all compared addresses will be output):

CON:

If the two files were identical, the A: prompt will appear.

If the files were different in any way, the results should scroll up the screen in three columns showing the hexadecimal address compared, the hexadecimal character of the first file found at that address and the hexadecimal character of the second file found at that address.

DEFAULTED FILE:

Identical - A: prompt will appear, and no .DIF file will be created.

Different - A: prompt will appear; using DIR command, .DIF file will be on disk and can be viewed using the TYPE command.

LST:

Identical - A: prompt will appear; no output to the printer.

Different Printer should begin printing three columns similar to CON output.

AUX:

Identical - A: prompt will appear; no output to serial device.

Different - Device should be activated with output similar to CON.

In this example, the two files were of different sizes. When you finally reach the bottom of the output, the word LARGER will appear under the file which was the larger of the two.

APPENDIX - J
SERDRV.OBJ Interrupt Driven Serial Port Routine

Serial port drivers for the Chameleon computer
Written by : Bob Depelteau
Copyright : Bob Depelteau 1983
Edit date : 9/21/83

DESCRIPTION

To use these routines, link SERVDRV.OBJ and CALL SERIALIO with AH and AL set for desired function.

INITIALIZE

(AH) = 0 Initialize a communications port
(AL) = Parameters for initialization

BAUD RATE	PARITY	STOPBIT	WORDLENGTH
765	43	2	10 BITS
000 - 110	00 - None	0 - 1	00 = 5 BIT
001 - 150	01 - Odd	1 - 2	01 = 6 BIT
010 - 300	11 - Even		10 = 7 BIT
011 - 600			11 = 8 BIT
100 - 1200			
101 - 2400			
110 - 4800			
111 - 9600			

(BX) = Length or transmit buffer
(CX) = Length of receive buffer
(Maximum size of any buffer is 64 K or hex 0000)
(Length assumed to be on 16 byte boundary)
(No check is made for out of memory)
(ES) = Starting Segment to use for buffers
Public variables:

```
RXBUFFSEG   DW  ?     ; Segment of receiver buffer
TXBUFFSEG   DW  ?     ; Segment of transmitter buffer
```

NOTE: Buffers are flushed and all information is lost upon initialization

TRANSMIT

(AH) = 1 Cue up a character in buffer to transmit
Updates tramsmitter buffer pointers
Public variables:

```
TXBUFFSTRT   DW    ; Offset start of buffer
TXBUFFEND    DW    ; Offset end of buffer
TXLENGTH     DW    ; Length of buffer
```

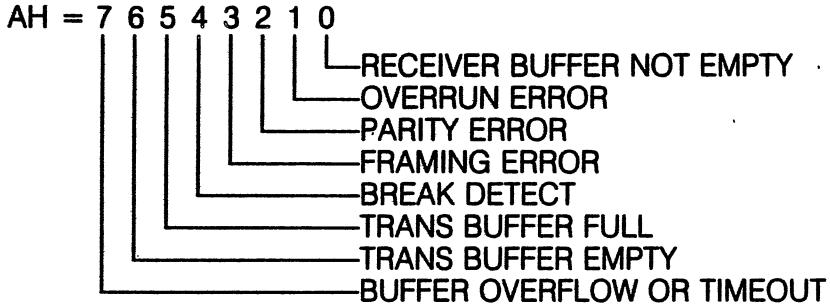
RECEIVE

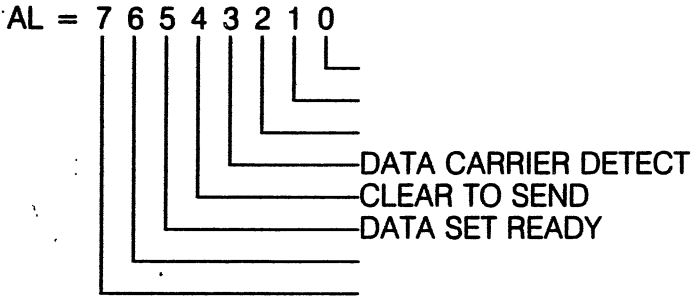
(AH) = 2 Get a character from buffer into (AL)
Updates Receiver buffer pointers
Public variables:

```
RXBUFFSTRT   DW    ; Offset start of buffer
RXBUFFEND    DW    ; Offset end of buffer
RXLENGTH     DW    ; Length of buffer
```

STATUS

(AH) = 3 Get port status into (AX)





TURN OFF INTERRUPTS

(AH) = 4 Close port communication
(Must be done before exiting program)

USED ON ALL ON EXIT

(AH) = Status of port as returned by function 3.No other registers are affected

APPENDIX - K
CHAMELEON BASIC86

MICROSOFT BASIC86 Rev 5.28
Sept. 6, 1983

CHAMELEON BASIC86 NOTES

The following BASIC files are included on the second MS-DOS disk:

RANDOM.BAS
SCREEN.BAS
SDEMO.BAS

RANDOM is a test program provided by MICROSOFT which demonstrates disk file I/O.

SCREEN is a set of subroutines for Clear Screen and Positioning the Cursor. It is set up to be added to you programs using the MERGE command under BASIC86. It consists of only subroutines and a program and therefore will not run by itself.

SDEMO is a program demonstrating screen controls using the subroutines provided in SCREEN.BAS.

APPENDIX - L
CHAMELEON OPTION PROGRAM

The OPTION program resident on the MS-DOS diskette is a utility designed to allow you to alter the hardware configuration of the Chameleon without having to physically change dip-switches or jumpers, as is the case with most other microcomputers. The program modifies registers in non-volatile RAM making the changes semi-permanent. The functions available under the OPTION program are shown below. This section contains a brief description of just what each function is, and what the result of invoking each is.

The CHAMELEON – OPTION MENU

A = Set System RAM Size	G = Set Keyboard Tone
B = Set Number of Printers	H = Set Serial Port A
C = Set A/D Port Option	I = Set Color Palettes
D = Set Initial Video Mode	J = Redirect Line Printer
E = Set Auto Boot Mode	K = Reset All Options
F = Set Number of Serial Ports	X = Exit Option Program

? = HELP !!!

ESC Key = Make No Change and Return to This Menu

ENTER CHOICE :

© Copyright 1983 SEEQUA Computer Corp. Version 2.3

BRIEF DESCRIPTION OF EACH FUNCTION

A. Set System Ram Size

This allows you to configure your machine to whatever RAM size is appropriate for your application. The default is 128 K. In order to choose RAM sizes above 128 K, the actual RAM chips must be present on the motherboard, i.e., you can't tell the computer it has 256 K if only 128 K is present on the motherboard. Maximum RAM currently possibly is 640 K bytes.

B. Set Number of Printers

This option allows for the use of multiple printers for the printing of output. The maximum number of printers supported is three (3). The default value is one (1).

C. Set A/D Port Option

This option allows you to tell the computer whether or not the A/D port is attached. This port is typically used for joysticks and game paddles. The default is the A/D port not attached.

D. Set Initial Video Mode

This is an extremely important function, and one typically exercised most often by the user. You have the ability to configure the video output to be either 40 column or 80 column, black & white or color, for a total of four (4) different video mode configurations.Because of the nature of the Chameleon architecture, if the brightness is turned up, IBM-PC programs that utilize color will generate green vertical bars on the internal video display. This is okay if you have an external color monitor that is showing the desired colors. However, if no color monitor is available, it could be annoying. Also, color text and graphics displayed on a black & white screen can sometimes be illegible. What is really desired in that case is a black & white representation of the color output that would eliminate the color bars on the monochrome screen. This can be accomplished by choosing the 80 column, black & white only option under this function. This creates a very nice gray scale when color is attempted.The black & white option is a good way to get crisp text and the highest clarity for all programs if a color monitor is not available. The black & white and color options are required if color output is desired through an external color monitor.

E. Set Auto Boot Mode

This function allows you to enable or disable the auto boot capability. The auto boot is what allows the Chameleon to proceed directly to load the operating system when powered on with a DOS diskette in Drive A. All internal diagnostics are run, and the "monitor" mode is completely bypassed. This is the default state of the system.By disabling the auto boot, you will always jump to the ROM monitor mode when initially powering up, or doing a warm boot. At this point, a **B** ⟨CR⟩ must be input in response to the cursor in order to continue booting into the operating system.

F. Set Number of Serial Ports

This option allows you to configure the system for up to seven (7) serial ports. The default number of serial ports is one (1)--the standard number of serial ports on a Chameleon.

G. Set Keyboard Tone

This function allows you to configure the intensity and pitch of the audio feedback when a keyboard key is depressed. The default is four (4) which is about medium. The audio feedback can be totally disabled by choosing zero (0). The range is 0 to 9.

H. Set Serial Port A

This function allows you to configure the parameters relating to the serial port (baud rate, word length, parity, number of stop bits). This is critically important when using the serial port as a communications port (through a modem, over a telephone line to a mainframe or mini). The settings are determined by the hardware that the serial port is connected to (the modem, the telephone line, and the mainframe/mini). Consult the manual for the hardware (mainframe and modem) you are connecting to. It is also important that the serial cable be configured properly for the device you are using.

I. Set Color Palettes

This option allows you to redefine the colors that are output to an external color monitor in graphics mode. Thus, you can make the sky appear green if you so desire. Obviously, this function has no use if an external color monitor is not employed, or if software that has color capabilities is not utilized. Any of the sixteen (16) available colors can be utilized.

J. Redirect Line Printer

This function allows you to automatically redirect printer output through the serial port (the default is the parallel port). This is very useful if you have a serial printer and wish to have the computer hardware know that printed output is to go through the serial port (as opposed to altering the software).

K. Reset All Options

This function allows you to reset all the above functions back to their default values. Chameleon Plus owners should be careful to remember that the system RAM size will default to 128 K and that Option A should be reset to 256 K immediately after using Option K.

X. Exit Option Program

This function must be invoked whenever changes have been made to any of the Option program functions. This will start the rewrite of the non-volatile RAM (where all the function parameters are permanently held until altered again).

? HELP !!!

This function will display a short help menu on the screen as a quick refresher or for those of you who are totally at a loss for what to do with the Option program functions.

INDEX

A

Alternate key 2-9
ASCII codes A-2
Assembly language B-1
Auto,
  command 8-5
  stopping 8-6
Aux I-2

B

Backup copies 6-1, 7-2
BASIC86 notes K-1
BIOS G-1
Booting disk 5-1
  Cold start 5-1
  Warm start 5-1
Branching, BASIC 8-9
Brightness control 3-4, 4-2
Buffer, CTERM H-4

C

Cabling F-1
Caps Lock 2-8
Care & Cleaning,
  Chameleon 3-5
  CRT 3-4
  disk drives 3-3
  keyboard 3-4
Carrying handle 4-3
Cold start 5-1
Color graphics 2-10
Color palettes L-4
Command,
  Auto 8-5
Communications package H-1
Comparing disks 5-8
Computer System 2-1
CON I-2

Control key 2-9
CPU,
Intel 8088 2-2
Zilog Z80A 2-2
Creating programs, BASIC 8-3
CRT 2-10
CTERM H-1
Cursor 2-8

D

Date, Entering system 5-2
Debugging, definition 8-7
Default drive 5-5
Delete key (Del) 2-10
Diskette,
care 3-1
copying 5-6
density 2-6
destination 5-7
directory (DIR) 5-11
diskcopy function 6-2, 7-3
inserting 3-2
labelling 6-3, 7-3
slave 7-3
source 5-7
storage 2-5
write protect 3-3
Disk Drives 2-5
Dot Matrix, screen 2-10
Drive specifier 5-11

E

Enter key 2-9
Escape key 2-9
Exiting BASIC 8-3
Expansion box 3-7
Extensions, file 5-11

F

Fan 2-5
FILCOM I-1

Files,
comparing I-1
maximum 5-10
File extensions 5-11
names 5-12
Filenames 5-10
assigning 5-13
Floppy Disks 3-1
(see Diskette)
Formatting disks 6-2, 7-3
Function keys,
(F1-F10) 2-10
CTERM H-2
Perfect Calc 7-7
Perfect Writer 6-9

G

Global characters 5-14
Graphics,
Color 2-10
High resolution 2-10
Ground, computer 3-6

H

High resolution graphics 2-10

I

Initial program load (IPL) 5-1
Insert key (Ins) 2-10
Inserting diskettes 3-2
Instruction set, B-2
8088 .....
Z80A B-5
Intel 8088 2-2
Interface chips 2-4
Interrupts G-3
I/O ports C-1

J,K

Keyboard 2-7
LED 2-7

plug 4-2
Keys 2-7
Alternate 2-9
Caps Lock 2-7
Control 2-9
Delete (Del) 2-10
Enter 2-9
Escape 2-9
Function (F1-F10) 2-10
Insert (Ins) 2-10
Numeric Lock 2-8
Print Screen 2-10
Return 2-9
Scroll Lock 2-10
Tab 2-9

L

Labelling diskettes 6-3, 7-3
Lessons,
Perfect Calc 7-7
Perfect Writer 6-8
Listing programs, BASIC 8-4
Loading
BASIC 8-1
software 5-1
LST I-2

M

Maximum files 5-10
Memory,
mapping D-1
size 2-3
Microprocessor 2-2
Monitor G-1
Motherboard 2-2
MSDOS 5-1
files 5-10
function keys 5-5
keys 5-5

N

Numeric Lock key 2-8
Numeric Pad 2-8

O

Operating systems 2-12
  installation 6-1
OPTION L-1
Optional interfaces 2-4
Optional equipment, care 3-7

P

Packing computer 4-4
Perfect Calc 7-1
  files 7-3
  overlay 7-3
Perfect Writer 6-1
  files 6-2, 6-4
  Menu 6-4
Ports 2-4
Port options L-2
Power,
  switch 4-3
  supply 2-5, 3-4
Print Screen key 2-10
Printing,
  Perfect Calc models 7-6
  Perfect Writer files 6-6

Q

Quitting,
  Perfect Calc 7-6
  Perfect Writer 6-5, 6-8

R

RAM 2-3
  parity checking 2-3
Rear panel (diagram) 4-2
Return key 2-9
ROM 2-3

S

Saving,
- BASIC programs 8-7
- Perfect Calc models 7-6
- Perfect Writer files 6-6

Scroll Lock key 2-10
Serial port interrupt J-1
Set up, computer 4-1
Slave disks 7-3
Software,
- definition 2-11, 3-6
- problems 3-7
- standard 2-11

Source disk 5-6
Special file characters 5-13
Spelling Checker 6-5
Special keys,
- Perfect Calc 7-9
- Perfect Writer 6-10

Status line,
- Perfect Calc 7-4
- Perfect Writer 6-6

T

Tab key 2-9
Time, entering system 5-3
Tone, keyboard L-4

U, V

Unpacking 1-1
Video display 2-10
Video mode L-3

W

Warmstart 5-1
Write protect, disk 3-3

X,Y,Z

Zilog Z80A 2-2

Microsoft MS™-DOS Disk Operating System

MS™-DOS user's guide

Information in this document is subject to change without notice and does not represent a commitment on the part of Microsoft Corporation. The software described in this document is furnished under a license agreement or non-disclosure agreement. The software may be used or copied only in accordance with the terms of the agreement. It is against the law to copy the MS-DOS Disk Operating System on cassette tape, disk, or any other medium for any purpose other than the purchaser's personal use.

Copyright © Microsoft, Corporation, 1982

LIMITED WARRANTY

MICROSOFT CORPORATION shall have no liability or responsibility to purchaser or any other person or entity with respect to any liability, loss or damage caused or alleged to be caused directly or indirectly by this product, including but not limited to any interruption of service, loss of business or anticipatory profits or consesquential damages resulting from the use or operation of this product. This product will be exchanged within twelve months from date of purchase if defective in manufacture, labeling or packaging, but except for such replacement the sale or subsequent use of this program is without warranty or liability.

THE ABOVE IS A LIMITED WARRANTY AND THE ONLY WARRANTY MADE BY MICROSOFT CORPORATION. ANY AND ALL WARRANTIES FOR MERCHANTABILITY AND/OR FITNESS FOR A PARTICULAR PURPOSE ARE EXPRESSLY EXCLUDED.

To report software bugs or errors in the documentation, please complete and return the Problem Report at the back of this manual.

MS-DOS is a trademark of Microsoft Corporation.
XENIX is a trademark of Microsoft Corporation.
UNIX is a trademark of Bell Laboratories, Inc.
CP/M is a trademark of Digital Research, Inc.

Document No. 8410-110-01
Catalog No. 1406
Part No. 14F07A

Microsoft

Welcome to the Microsoft family of products.

The Microsoft Corporation is the recognized leader in microcomputer software. The Microsoft BASIC interpreter, in its several versions, is the standard high-level programming language for microcomputers. The Microsoft Corporation provides other software of consistently high-quality that sets the standard for software quality.

In addition to the MS-DOS Disk Operating System and the Microsoft BASIC interpreter, Microsoft sells other full-feature language compilers, operating systems, and utility software. For all this software, Microsoft strives to offer a family of software products that both look alike from one product to the next, and can be used together for effective program development.

For more information about other Microsoft products, contact:

Microsoft Corporation
The Microsoft Building
10700 Northup Way
C-97200
Bellevue, WA 98004
(206) 828-8080

CONTENTS

Chapter 1 INTRODUCTION
1.1 Features and Benefits of MS-DOS 1-1
1.2 Provided Software 1-4
1.3 System Start-Up 1-6
1.4 Syntax Notation 1-7

Chapter 2 SYSTEM STRUCTURE
2.1 System Resources 2-1
2.2 File System 2-4
2.3 User Interface 2-8
2.4 Command Types 2-16

Chapter 3 COMMANDS
CHKDSK 3-3
COPY 3-5
DATE 3-9
DEL 3-10
DIR 3-11
EXE2BIN 3-12
FORMAT 3-14
PAUSE 3-15
REM 3-16
REN 3-17
SYS 3-18
TIME 3-19
TYPE 3-20

Chapter 4 EDLIN
4.1 Invocation 4-1
4.2 Intraline Commands 4-3
4.3 Interline Commands 4-14
4.4 Error Messages 4-36

Chapter 5 DEBUG
5.1 Invocation 5-1
5.2 Commands 5-2
5.3 Error Messages 5-35

Chapter 6 FILCOM
6.1 Invocation 6-2
6.2 Commands 6-3
6.3 Examples 6-10

Appendix A Instructions for Single Disk Drive Users
Appendix B The File Control Block
Appendix C Interrupts and Function Calls
Appendix D Disk Errors

CHAPTER 1

INTRODUCTION

The MS-DOS disk operating system is one of Microsoft's family of operating systems for 8086 and 8088 microprocessors. It provides a simple but powerful interface between the user and a computer system's resources. Most all Microsoft languages are available under MS-DOS, including the BASIC Interpreter, the BASIC Compiler, MS-Pascal, and MS-FORTRAN. In addition, the 8-bit versions of Microsoft's languages are upward compatible with the 16-bit versions. Thus, application programs written in 8-bit Microsoft languages can be run under MS-DOS with little or no modification.

1.1 FEATURES AND BENEFITS OF MS-DOS

The following features and benefits make MS-DOS the operating system of choice for 8088 and 8086 microcomputers:

Easy Conversion from 8080 to 8086

MS-DOS allows as much transportability of 8-bit machine language software as is reasonably possible. For instance, MS-DOS emulates system calls to the 8-bit CP/M operating system. Therefore, by simply running assembly language source code through the Intel conversion program, almost all 8080 programs created for the CP/M operating system can be made to work without modification in the MS-DOS environment. In most cases, converting programs from CP/M-80 to MS-DOS is easier than converting to other 16-bit operating systems.

Device Independent I/O

MS-DOS simplifies I/O to different peripheral devices by assigning a reserved filename to each device. These names are built-in to MS-DOS and are detected by the MS-DOS file system. Thus, for example, programs designed only for disk file I/O

can have their input come from the terminal keyboard or their output sent to the printer.

Advanced Error Recovery Procedures

MS-DOS does not necessarily require rebooting when disk errors occur. If a disk error occurs at any time during any program, MS-DOS retries the operation three times. If the operation cannot be completed successfully, MS-DOS returns an error message, then waits for the user to enter a response. Thus, the user can attempt to recover from the error rather than reboot the operating system.

Complete Program Relocatability

The architecture of the 8086 CPU limits each segment of memory to 64K-bytes and requires intersegment references to be fixed for a given load address. MS-DOS works around this limitation through its special executable object module format. During program development, the Microsoft linker can combine object modules created with any of Microsoft's BASIC, Pascal, or FORTRAN compilers or by Microsoft's macro assembler. These modules can be combined to create an executable module requiring any number of segments.

Powerful, Flexible File Characteristics

In MS-DOS, there is no practical limit on file or disk size. MS-DOS uses 4-byte XENIX Operating System compatible logical pointers for a disk capacity of up to 1 gigabyte. (Microsoft's XENIX is a licensed version of the UNIX system. XENIX is available for a variety of 16-bit microcomputer systems through a license agreement from Microsoft.)

MS-DOS remembers the exact physical end-of-file marker. Thus, should one open a file with a logical record length greater than the physical record length, MS-DOS remembers exactly where the file ends to the byte, rather than rounded to 128 bytes.

Written entirely in 8086 Assembly Language

Because it is written entirely in 8086 assembly language, MS-DOS provides significant speed improvements over operating systems that are largely translated from their 8-bit counterparts.

Fast, Efficient File Structure

MS-DOS employs a highly efficient disk structure which eliminates the need for "extents," minimizes access to the directory track, and provides for duplicate directory information and verifications

after writes.

No Need to Log in Disks
As long as no file is currently being written, disks can be swapped without logging in. One benefit of this feature is that the MS-DOS debugger, DEBUG.COM can be used without reloading for different programs on different disks.

No Physical File/Disk Size Limitation
Unlike operating systems that are limited to 8-megabytes per disk, MS-DOS does not require breaking a 24-megabyte hard disk into three separate logical drives.

No Overhead for Non-128-Byte Physical Sectors
Since MS-DOS does its own blocking and deblocking of disk sectors, there is no reason to worry about different physical sector sizes when writing the low level routines for a particular computer system.

Time and Date Stamps
When a file is modified, MS-DOS automatically records the time and date of the modification.

100% IBM Compatible
International Business Machines Corporation has chosen MS-DOS (called IBM Personal Computer DOS) to be the preferred operating system for the IBM Personal Computer. IBM has already announced Microsoft BASIC, Pascal, and FORTRAN along with other accounting, financial planning, and word processing software that runs under MS-DOS.

1.2 PROVIDED SOFTWARE

The software provided with the MS-DOS operating system is described below:

CHKDSK.COM

CHKDSK.COM is a command used to check and verify the contents of a disk. It is further described in Chapter 3, "Commands."

COMMAND.COM

COMMAND.COM is the command interpreter used to interface between the user and the underlying operating system. It allows the user to perform file management functions such as rename and delete, as well as to load and execute programs. COMMAND.COM is further described in Chapter 2, "System Structure."

CREF.EXE

CREF.EXE is the Microsoft MS-CREF cross-reference utility used to create a cross-reference listing from an assembly source listing. CREF.EXE is further described in the Utility Software Package Manual.

DEBUG.COM

DEBUG.COM is a debugger program used to provide a controlled testing environment for executable object files. DEBUG.COM is further described in Chapter 5, "DEBUG."

EDLIN.COM

EDLIN.COM is the MS-DOS line editor. Intraline editing is performed using the special editing keys that are also available at the MS-DOS command level. EDLIN.COM is further described in Chapter 4, "EDLIN."

EXE2BIN.COM

EXE2BIN.COM is used to convert .EXE files to .COM files. In general, only assembly language programs that have been specially formulated may undergo such conversions. EXE2BIN.COM is further described in Chapter 3, "Commands."

FORMAT.COM

FORMAT.COM is used to format disks so that they can be used with MS-DOS. FORMAT.COM is described in Chapter 3, "Commands."

FILCOM.COM

FILCOM is a file comparison program used to check for differences between files. Either text or binary files may be compared. FILCOM is further described in Chapter 6, "FILCOM."

IO.SYS

IO.SYS is the lowest level of the MS-DOS operating system, interfacing to all I/O devices. It is an MS-DOS "hidden file" and does not show up when a directory command is executed. IO.SYS is automatically loaded into memory when your system is booted up. See Chapter 2, "System Structure" for more information.

LIB.EXE

LIB.EXE is the Microsoft MS-LIB library manager used to create, maintain, and manipulate libraries of object files. LIB.EXE is further described in the Utility Software Package Manual.

LINK.EXE

LINK.EXE is the Microsoft MS-LINK linker used to link object files and object libraries to create executable .COM and .EXE files. LINK.EXE is further described in the Utility Software Package Manual.

MASM.EXE

MASM.EXE is Microsoft's relocatable macro assembler for 8086 and 8088 microprocessors, MACRO-86. MASM.EXE is further described in the Utility Software Package Manual.

MSDOS.SYS

MSDOS.SYS is the heart of the MS-DOS operating system, where most management of system resources takes place. MSDOS.SYS is intimately tied to COMMAND.COM and IO.SYS. Note that MSDOS.SYS is an MS-DOS "hidden file" and does not show up when a directory command is executed. MSDOS.SYS is automatically loaded into memory when your system is booted up. For further information, see Chapter 2, "System Structure."

SYS.COM

SYS.COM is used to transfer MSDOS.SYS and IO.SYS from a system disk to a formatted disk that does not contain the MS-DOS operating system on it. It is further described in Chapter 3, "Commands."

1.3 SYSTEM START-UP

To start-up your system, follow your manufacturer's instructions. Next, insert your system disk in drive A:. At this point, your MS-DOS will be booted up and loaded from disk into program memory. A banner then appears containing the MS-DOS version number. Next, COMMAND.COM is loaded into memory and a banner containing it appears. Then you are prompted to set the date and time with the DATE and TIME commands.

For example, you might type:

```
Current time is 00:00:00.00
Enter new time:10:30
Current date is 1-1-80
Enter new date:3-25-82
```

Finally, the MS-DOS prompt appears with the letter of the driver and a colon:

```
A:_
```

The cursor is indicated in this manual with an underline character. At this point you are at the MS-DOS command level and are under the supervision of COMMAND.COM.

The first thing that you should do is create a back-up of your system disk. This is done by first inserting a blank disk into drive B:. Next type:

```
FORMAT B:/S
```

FORMAT formats the disk in drive B: so that it can be used

with MS-DOS. The /S causes hidden system files to be copied to the newly formatted disk after it has been formatted. When FORMAT is done, you should then type:

```
COPY A:*.* B:
```

This command copies all files on the system disk to the new disk. At this point, you should remove the original system disk and store it in a safe place. Use the copy of the system disk from now on.

1.4 SYNTAX NOTATION

The following notation is used throughout this manual in descriptions of command and statement syntax:

[ ] Square brackets indicate that the enclosed entry is optional.

< > Angle brackets indicate user entered data. When the angle brackets enclose lower case text, the user must type in an entry defined by the text; for example, <filename>. When the angle brackets enclose upper case text, the user must press the key named by the text; for example, <RETURN>.

{ } Braces indicate that the user has a choice between two or more entries. At least one of the entries enclosed in braces must be chosen.

... Ellipses indicate that an entry may be repeated as many times as needed.

CAPS Capital letters indicate portions of statements or commands that must be entered exactly as shown. When capital letters appear within angle brackets, they indicate typing of a control character such as <CONTROL-C> or <RETURN>.

All other punctuation, such as commas, colons, slash marks, and equal signs, must be entered exactly as shown.

CHAPTER 2

SYSTEM STRUCTURE

This chapter gives a structural overview of the MS-DOS operating system, describing:

1. System resources
2. The file system
3. The user interface
4. Command types

By acquiring an understanding of each of these subjects, you will gain a sound understanding of the structure of the MS-DOS operating system.

2.1 SYSTEM RESOURCES

Each time your computer is turned on, it normally will "boot up" the operating system by automatically loading MS-DOS from disk into memory. The area of memory in which the operating system is loaded is referred to as system memory.

MS-DOS, itself, consists of three files:

```
COMMAND.COM
MSDOS.SYS
IO.SYS
```

These three files combine to form an operating system that controls all system resources. Note that MSDOS.SYS and IO.SYS are "hidden" files that are not displayed when a directory command is executed. The relationship between these three files and system resources is shown in Figure 2.1.

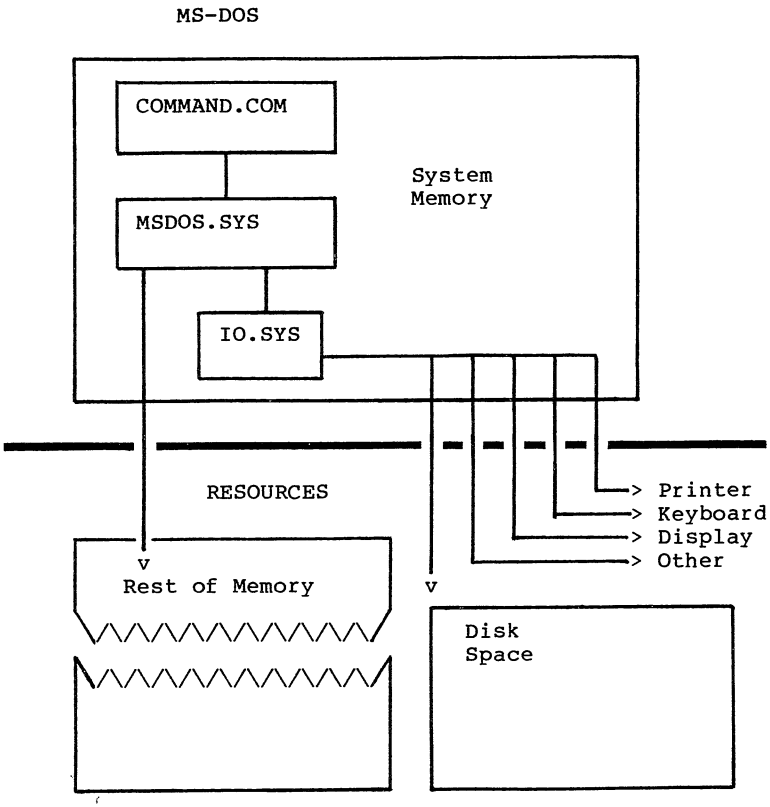


Figure 2.1 MS-DOS and Its Resources

System resources include peripheral devices such as terminals, printers, and serial lines. However, a system's most important system resources are its disk space and its memory--these are described in the following sections.

2.1.1 Disk Space

In MS-DOS, disk space is divided into four parts:

The Reserved Sectors

This region contains information that is used each time MS-DOS is booted up. This information is system dependent, but typically will include a simple bootstrap loader.

The Directory

The directory contains information about each file on a given disk, including the file's complete filename, its size, and its time and date of last modification.

The File Allocation Table

The file allocation table (FAT) contains location information for the data making up each file on a given disk. Note that MS-DOS does not require a file's contents to reside in physically contiguous disk sectors.

Files

The great majority of disk space is reserved for the contents of files. An individual file does not necessarily reside in contiguous sectors on disk, and may be "scattered" in memory to decrease waste of disk space.

2.1.2 Memory

Besides controlling a system's disk space and its other devices, MS-DOS must also control main memory. This means that MS-DOS must be capable of loading files into memory either as data files or as files that are to be executed. The actual loading of files is performed by IO.SYS, the lowest level of the MS-DOS operating system. Loading of executable files is supervised by COMMAND.COM. For most well-designed programs, control is returned to MS-DOS after either normal or abnormal termination of a program.

Note that part of COMMAND.COM may be overlaid to make room for a particularly large executable file. After execution of such a file, MS-DOS automatically loads the overlaid part of COMMAND.COM back into system memory, and normal execution of COMMAND.COM resumes. If the overlaid part of COMMAND.COM is not available on disk because the disk on which it resides has been removed, the following message appears:

```
Insert DOS disk in default drive
and strike any key when ready
```

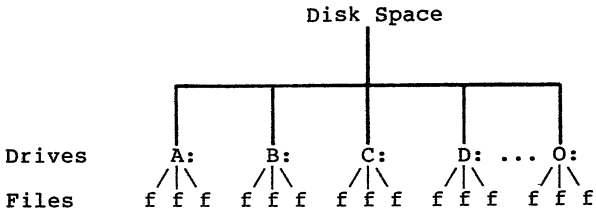
Also, if an incorrect version of COMMAND.COM is found, then a similar message appears:

```
Invalid COMMAND.COM
Insert DOS disk in default drive
and strike any key when ready
```

2.2 FILE SYSTEM

The preceding discussion of system resources discussed many of the internal aspects of the operating system. A file system, on the other hand, can be thought of as the external organization of system resources. It provides a way of talking about files and devices. Note that MS-DOS supports "device independent I/O", which means that the distinction between files and devices is an internal distinction, but not an external one. Therefore, the user can treat files and devices alike, and can refer to either with "filenames."

Note, however, that disk space is special, since it is divided into drives; the disk space in a drive and on a particular disk is further divided into files, as shown below:



Disks are named so that up to 15 can be referred to. They are named with the letters A through O, where each letter is followed by a colon (:). This colon separates the name of the disk from the names of individual files on the disk. This letter-colon combination is called a drive designation.

2.2.1 Naming Conventions

MS-DOS supports a three-part name for files (and devices) called a file specification. A full file specification contains a drive designation, a filename, and a filename extension.

The format of a file specification is:

```
[<d>:]<filename>[.<ext>]
```

The parts of a file specification are described below:

<d>: is the drive designation as described in the preceding section.

<filename> is an internal name consisting of from 1 to 8 characters. Internally, all filenames are exactly 8 characters: this means that filenames with seven or fewer characters have all remaining characters padded with spaces. MS-DOS performs this padding automatically. Legal characters in filenames are:

```
A-Z   0-9   $   &   #   @   !

%   '   (   )   -   <   >

\   ^   {   }   ~   |   `
```

Any of the above characters is legal in any position in the filename. Note that lowercase characters are converted to uppercase wherever they occur in a file specification. This means that a filename such as "FiLe.ExT" is converted to the filename "FILE.EXT."

.<ext> is the filename extension consisting of three or fewer characters. Padding of spaces occurs as described above for filenames. Legal characters are the same as for filenames. All characters are legal in any position in the extension, and all lowercase characters are converted to uppercase characters, just as for filenames. Filenames with no extensions can also be specified by typing only a period after the filename. It is usually only necessary to use this form when a given command requires a default extension.

2.2.2 Wild Card Characters

There are two wild card characters that can be used in file specifications: the asterisk (*) and the question mark (?). By using these characters, a short hand notation is created for specifying multiple files. This notation is particularly useful when file specifications are required as parameters for commands. MS-DOS makes full use of this capability for commands such as the directory, delete, and copy commands: DIR, DEL, and COPY.

The two wild card characters are described below:

? When COMMAND sees a question mark (?), it matches any single character found in that position in a filename.

For example, examine the following command:

```
DEL AB?DE.EXT
```

This directory command deletes all files whose names begin with AB, end with DE, and have the filename extension .EXT. For example, MS-DOS might delete the following files:

```
ABCDE.EXT
ABODE.EXT
ABIDE.EXT
```

* When COMMAND.COM sees an asterisk (*) in a filename parameter, it matches all characters found in those positions in any filename in the appropriate directory. The asterisk (*) is a shorthand for a series of question marks (?). So, for example, the following are equivalent:

```
DEL *.*
DEL ???????.???
```

However, the asterisk is typed more easily.

The command DEL *.* deletes all files stored on the disk in the default drive. regardless of filename or extension.

Here are some other examples, with their equivalents in question mark characters:

```
DEL FILE.*     DEL FILE.???
(Delete all variations of FILE
 regardless of extension)
```

```
DEL *.EXT      DEL ????????.EXT
(Delete all files with the
 filename extension .EXT)

DEL ABC*.E*    DEL ABC?????.E??
(Delete all files whose names
 begin with ABC and that have
 a filename extension that
 begins with .E)
```

2.2.3 Device Filenames

Certain 3-letter filenames are reserved for the names of devices. These names are listed below:

AUX
: Used when referring to input from or output to an auxiliary device.

CON
: Used when referring to either keyboard input or to output to the terminal screen.

LST or PRN
: Used when referring to the line printer.

NUL
: Used when you do not want to create a particular file, but the syntax of a command requires an input or output filename.

Even if given device designations or extensions, these filenames remain associated with the devices listed above. Thus, A:CON.LST still refers to the terminal console and is not the name of a disk file. This device naming scheme permits treating devices as if they were files, and is a consequence of MS-DOS's device independent I/O.

2.3 USER INTERFACE

MS-DOS acts as an interface between the user and a computer system's resources, with communication from the user normally occurring through keyboard input. This keyboard input is the raw data that is used to edit the MS-DOS command line. When editing is complete, the command line is passed on to COMMAND.COM where it is scanned for command names and parameters. Thus, the user interface consists of two levels of processing: command line editing and command interpretation. These levels are discussed in the next two sections.

2.3.1 Command Line Editing

MS-DOS offers a variety of functions that operate on the command line buffer. These functions make command line editing a simple and efficient task, very much in contrast to the nuisance it can be in other operating systems. The command line buffer is intimately related to another buffer called the "template," which is used in many of MS-DOS's special editing functions.

The model for command line input is as follows:

1. When text is entered from the keyboard, it is held in the command line until the <RETURN> key is pressed.

2. Pressing <RETURN> causes the contents of the command line to be sent to COMMAND.COM for processing.

3. Pressing <RETURN> also copies the command line to the template.

Thus, the template always contains the last entered command line.

The command line is altered by entering one the following kinds of input:

Alphanumerics
Punctuation
Special editing functions
Control character functions

Alphanumeric and punctuation characters are entered into the command line as they are typed. Later, COMMAND.COM will convert all lowercase letters to uppercase.

The special editing and control character functions greatly increase MS-DOS's ease of use. All of the MS-DOS commands, from DEBUG to FILCOM, can make use of these functions wherever input is required from a terminal. These functions are always resident in the operating system.

2.3.1.1 Special Editing Functions - Because they depart from the "normal" way in which most operating systems handle command input, the special editing functions deserve particular emphasis. They relieve the user of repeatedly typing in the same sequences of keys because the last command line entered is automatically placed in the template and "remembered."

Therefore, by using the template and executing the special editing functions, you receive the following advantages:

1. A command line can be instantly repeated in two key-strokes.

2. An erroneous command line can be edited and retried, without reentering the entire command line.

3. A command line similar to a preceding command line can be edited and executed with a minimum of typing.

The relationship between the command line and the template is shown in Figure 2.2.

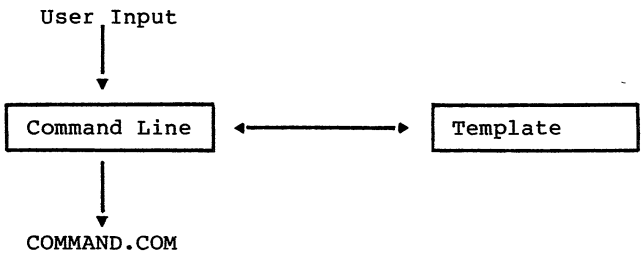


Figure 2.2 The Command Line and the Template

Table 2.1 contains a complete list of the special editing commands. Each command is more fully described in Chapter 4, "EDLIN," where these special editing commands become a subset of the commands available within the MS-DOS editor, EDLIN, and are called the intraline editing commands.

Note that the special editing commands may be assigned to the keys that make the best use of a specific terminal keyboard. Therefore, each command is identified by a functional name rather than by a specific key code. For an application on a specific terminal, the codes are configured for specific terminal keys.

Table 2.2 Special Editing Functions

Key	Editing Function
<C1>	Copy one character from the template to the command line
<CM>	Copy all characters up to the character specified from the template to the command line
<CT>	Copy all remaining characters in the template to the command line
<S1>	Skip over (do not copy) a character in the template
<SM>	Skip over (do not copy) the characters in the template up to the character specified
<QI>	Void the current input; leave the template unchanged
<INS>	Enter insert mode
<REP>	Exit insert mode (toggle from insert); this is the default
<NT>	Make the new line the new template

As an example of the use of the special editing keys and command entry in general, pretend that you have entered the following command:

```
DIR PROG.COM<RETURN>
```

This command displays contents of the file PROG.COM on the terminal screen. It also has the useful side effect of saving the command line in the template. To repeat the command, all you have to do is type two keys: <CT> and <RETURN>.

The repeated command is displayed on the screen, as you type, as shown below:

<CT>DIR PROG.COM<RETURN>

Note that pressing the <CT> key causes the contents of the template to be copied to the command line buffer; pressing <RETURN> causes the command line to be processed by COMMAND.COM.

Now pretend that you want to display the contents of the file PROG.ASM. To do this we will make use of the template, and type

<CM>C

Entering <CM>C copies characters from the template to the command line buffer up to the character "C":

DIR PROG._

Note that the underline is your cursor. Now type:

.ASM

The result is:

DIR PROG.ASM_

The desired command line is now in the command line buffer. To send this command line on to COMMAND.COM, the command interpreter, simply enter <RETURN>.

The template now contains the following command line:

DIR PROG.ASM

Now assume that we want to execute the following command:

TYPE PROG.ASM

To do this, we type:

TYPE<INS> <CT><RETURN>

Note that normal alphanumeric characters are entered directly into the command line buffer, automatically replacing corresponding characters in the template. This automatic replacement is turned off when the <INS> key is typed. Thus, the characters "TYPE" replace the characters "DIR " in the template. To insert a space between "TYPE" and "PROG.ASM," we first typed <INS> and then a space. Finally, to copy the rest of the template to the command line, we typed <CT>, followed by a <RETURN>.

When <RETURN> is typed, the entire command line is copied to the template, in this case:

```
TYPE PROG.ASM
```

If we had misspelled "TYPE" as "BYTE", a command error would occur. Still, instead of throwing away the whole command line, we could have saved the mistyped command line (before typing <RETURN>) by creating a new template with the <NT> key:

```
BYTE PROG.ASM<NT>
```

We could then edit this erroneous command line by typing:

```
T<Cl>P<CT>
```

The <Cl> key copies a single character from the template to the command line buffer. The resulting command line then is what we want:

```
TYPE PROG.ASM
```

As an alternative, we could have used the same template containing BYTE PROG.ASM, and used the <Sl> and <INS> commands to achieve the same result:

```
<Sl><Sl><Cl><INS>YP<CT>
```

To illustrate how the command line buffer is affected as you type, examine the keys typed on the left and their affect on the contents of the command line buffer, shown on the right:

```
<Sl>      _                  {Skips over 1st template char}
<Sl>      _                  {Skips over 2nd template char}
<Cl>      T_                 {Copies 3rd template char}
<INS>YP   TYP_               {Inserts two characters}
<CT>      TYPE PROG.ASM_     {Copies rest of template}
```

Note that <Sl>, like <SM>, does not affect the command line buffer; rather, it effects the template by deleting the first character in the template. Similarly, <SM> deletes characters in the template up to a given character (this character is the next one typed).

As you can see from the above examples, these special editing function keys can add greatly to your effectiveness at the keyboard. The next section describes control character functions that complement the above functions.

2.3.1.2 Control Character Functions - While commands are being entered, MS-DOS recognizes seven control character functions. These control characters and the functions associated with them are shown in Table 2.2.

Table 2.2 Control Character Functions

Control Character	Function
<CONTROL-N>	Cancel echoing of output to line printer.
<CONTROL-C>	Abort current command.
<CONTROL-H>	Remove last character from command line, and erase character from terminal screen.
<CONTROL-J>	Insert physical end-of-line, but do not empty command line. Use Linefeed to extend the current logical line beyond the physical limits of one terminal line.
<CONTROL-P>	Echo terminal output to the line printer.
<CONTROL-S>	Suspend display of output to terminal screen. Press any key to resume.
<CONTROL-X>	Cancel the current line, empty the command line, and then output a back slash (\), carriage return, and line feed. The template used by the Special Editing commands is not affected.

2.3.2 Command Interpretation

The MS-DOS user interface permits editing of command lines with the special editing and control character functions described in the preceding section. Once a command line has been edited, it is sent to COMMAND.COM for processing. COMMAND.COM is the hub of the operating system, acting as the interface between the lower levels of the operating system and user input. It is in COMMAND.COM that the commands that are entered on the command line are interpreted.

Commands themselves are of one of two types: either internal or external. Internal commands are all resident in memory as part of COMMAND.COM; they are loaded into memory when the operating system is booted up. External commands, on the other hand, are loaded into memory from disk only when needed. External commands reside in disk files that have a name with either a .COM or .EXE extension. Note that COMMAND.COM itself is an external command. A picture of the command interface is shown in Figure 2.3.

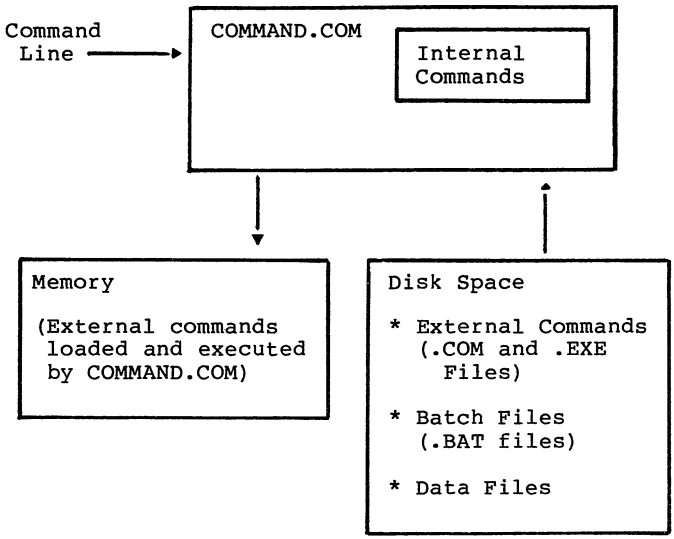


Figure 2.3 MS-DOS Command Structure

Command interpretation begins when COMMAND.COM scans the command line for the name of a legal command. If it is an internal command, it is executed immediately; if it is a batch file, commands are executed indirectly from a .BAT

file; if it is an external .COM or .EXE command, the appropriate file is loaded from disk into memory where it is executed. (The batch facility and .BAT commands are discussed later in this chapter.)

COMMAND.COM provides MS-DOS' characteristic colon prompt (:) in the form of a drive designation letter and a colon (:).

For example:

```
A:_
```

The cursor is the focus of any editing actions that you perform. The cursor is indicated by an underline in this manual; the symbol itself is implementation dependent.

At system start-up, the default prompt is always A:. After start-up, the user may specify the default (that is, the currently selected drive). To select a new drive, simply enter the designation letter followed by a colon:

```
A:_                 (prompt; default drive)
A:B:<RETURN>        (user enters new drive designation)
B:_                  (new prompt; new default drive)
```

Note that COMMAND's main goals are to identify commands typed at the command line and then to execute them. All commands consist of a command name followed by optional parameters. When parameters are present, they must be separated from the command name and from each other. Spaces, tabs, and commas are the only legal separators.

For example:

```
COPY OLDFILE.REL,NEWFILE.REL

RENAME THISFILE THATFILE
```

COMMAND.COM is able to execute several different types of commands: these types are described later in this chapter.

2.4 COMMAND TYPES

COMMAND.COM allows you to execute four different types of commands:

1. Internal commands such as DIR, REN, TYPE and DEL.
2. .COM commands such as CHKDSK.COM, DEBUG.COM, and EDLIN.COM.
3. .EXE commands such as LINK.EXE, MASM.EXE, and CREF.EXE.
4. .BAT command files that contain multiple instances of the above commands.

The internal commands and the simplest, most commonly used external commands are described in Chapter 3, "Commands." You should study all of these commands carefully.

Use of the MS-DOS batch facility is discussed in the next section of this chapter. The batch facility allows you to indirectly execute a set of commands contained in a file. This facility allows you to tailor commands for a particular purpose without a great deal of programming effort.

Separate chapters are also provided on three utility programs:

EDLIN - The MS-DOS line editor
DEBUG - The MS-DOS debugger
FILCOM - The MS-DOS file comparison program

Finally, most of the .EXE files provided with the MS-DOS operating system are described in the Utility Software Package Manual. The programs described in this manual make up a powerful and complete assembly language development system.

This software includes:

MASM.EXE - The MACRO-86 macro assembler for 8088 and 8086 microprocessors
LINK.EXE - The MS-LINK linker
CREF.EXE - The MS-CREF assembler cross-reference utility
LIB.EXE - The MS-LIB library manager facility

2.4.1 Internal Commands

Internal commands are incorporated in COMMAND.COM and are always available when COMMAND.COM is resident in memory. Unlike the external commands, discussed in the next section, these commands need not be available on disk when they are executed. Note that most internal commands are simple and easy to use. This is in contrast to some of the external commands which are larger and more complex.

The internal commands are listed below. Each is described thoroughly in the next chapter.

```
COPY        REN
DATE        PAUSE
DEL         TIME
DIR         TYPE
REM
```

2.4.2 External Commands

Any file with the filename extension .COM or .EXE is considered valid as an external command. Such commands are executed by entering the name of the file less its .COM or .EXE extension. Programs that you create with most languages will be .EXE files.
External commands include:

```
ASM86       EXE2BIN
CHKDSK      FORMAT
COMMAND     FILCOM
CREF        LINK
DEBUG       LIB
EDLIN       SYS
```

Note that .EXE files created with the MACRO-86 assembler can be converted to .COM files with the command EXE2BIN.EXE. The format of a .COM file is special, so .EXE files cannot be arbitrarily converted. Note also that all .COM commands execute in less than 64K of memory; .EXE files, on the other hand, may require more than 64K of memory to execute.

2.4.3 Batch Commands

The MS-DOS batch facility allows files containing commands to be submitted for processing internally by MS-DOS. "Batches" of commands in such files are processed as if they were typed at a terminal. Each batch file must be named with the .BAT extension, and is submitted for execution by entering its filename less that extension. Optional parameters may be given as well. Therefore, the invocation syntax is as follows:

```
<filespec>[<parameters>]
```

By creating a .BAT file with prototype commands containing positional parameters, parameters may be passed to the .BAT file when it is executed. You may specify up to 10 positional parameters, named %0 through %9.

Positional parameters are substituted in their order on the invocation line for corresponding occurrences in the batch file. If the dummy parameter %0 is used, the batch facility substitutes the name of the batch command itself for parameter %0. Thus, the batch facility permits creation of batch commands that can be used on more than just one set of files, and that can be used to reexecute themselves.

For example, a batch file might look like this when viewed from within EDLIN, the MS-DOS line editor:

```
1: REM This is file NEWDISK.BAT
2: REM   (the .BAT extension must be given)
3: PAUSE  Insert disk in B:
4: FORMAT B:/S
5: DIR B:
6: CHKDSK B:
```

To execute this .BAT file, simply enter the filename without the .BAT extension:

```
NEWDISK
```

The result is the same as if each of the lines in the .BAT file were entered at the terminal as individual commands.

To pass parameters to the .BAT file, the user must create a .BAT file containing prototype commands with dummy entries.

For example:

```
1: REM This is A:ASMFILE.BAT
2: REM START BATCH FILE
3: COPY %1.ASM %2.ASM
4: MASM %2,%2,%2;
5: TYPE %2.PRN
6: TYPE %0.BAT
```

Assume that this file exists as A:ASMFILE.BAT.

To execute this .BAT file and pass parameters, enter:

```
A:ASMFILE A:MYPROG B:MYPROG
```

The result is the same as if you had entered each of the following commands at your terminal:

```
REM This is A:ASMFILE.BAT
REM START BATCH FILE
COPY A:MYPROG.ASM B:MYPROG.ASM
MASM B:MYPROG,B:MYPROG,B:MYPROG;
TYPE B:MYPROG.PRN
TYPE A:ASMFILE.BAT
```

When you boot up your system, COMMAND.COM searches for the file A:AUTOEXEC.BAT. If a file with that name exists on disk, then the batch facility is automatically invoked to execute the commands contained in AUTOEXEC.BAT. In such a case, execution of the TIME and DATE commands at start-up is bypassed. If COMMAND.COM does not find AUTOEXEC.BAT, then the normal MS-DOS prompt is displayed instead.

Two MS-DOS commands are available expressly for their use in batch files: REM and PAUSE. REM permits the inclusion of remarks and comments in batch files; PAUSE prompts the user with an optional message and permits either continuing or aborting execution of a batch file at a given point. REM and PAUSE are further described in Chapter 3, "Commands."

CHAPTER 3

COMMANDS

NOTE

Users of single-drive systems should refer to Appendix A for the additional procedures required when executing many of the following commands.

The following notation is used in the descriptions in this chapter.

filespec Refers to an optional drive designation followed by a filename followed by a period and an optional three letter filename extension. For example:

```
B:ABODE.BAS   (refers to a disk file
                 on disk B:)
FILE.BAS      (refers to a disk file
                 on the default disk)
CON           (refers to the user's
                 terminal console)
CON.BAS       (same as above)
```

filename Refers to any valid name for a disk file, including an optional extension. A filename parameter does not refer to a device or to a disk drive designation alone.

d: Refers to a disk drive designation

All of the commands described in this chapter are listed below:

CHKDSK	Scan the directory of the default or designated drive and check for consistency.
COPY	Copy file(s) specified.
DATE	Display and set date.
DEL	Delete files specified. ERASE is a synonym for this command.
DIR	List requested directory entries.
EXE2BIN	Convert .EXE file to a .COM file.
FORMAT	Format a disk to receive MS-DOS files.
PAUSE	Pause for input in a batch file.
REM	Display a comment in a batch file.
REN	Rename first file as second file. RENAME is a synonym for this command.
SYS	Transfer MS-DOS system files from drive A: to the drive specified.
TIME	Display and set time.
TYPE	Display the contents of file specified.

NAME TYPE

CHKDSK External

SYNTAX

CHKDSK [d:]

FUNCTION

Scan the directory of the default or designated drive and check it for consistency.

COMMENTS

CHKDSK should be run occasionally on each disk to verify the integrity of the directory structure. If any errors are found, the appropriate error message is displayed and corrective action is attempted.

After the disk has been checked, CHKDSK displays error messages, if any, and then a status report.

A sample status report follows:

```
160256  bytes total disk space
  8192  bytes in 2 hidden files
 30720  bytes in 8 user files
121344  bytes available on disk

 65536  bytes total memory
 53152  bytes free
```

If an error is detected, CHKDSK returns one of the following error messages:

Allocation error for file <filename>
: The named file had a data block allocated to it that did not exist (that is, a data block number larger than the largest possible block number). CHKDSK truncates the file short of the bad block.

Disk not initialized
: No directory or file allocation table was found. If files exist on the disk, and the disk has been physically harmed, it may still be possible to transfer files from this disk to recover data.

Directory error-file: <filename>
: No valid data blocks are allocated to the named file. CHKDSK deletes the file.

Files cross-linked: <filename> and <filename>
The same data block is allocated to both files. No corrective action is taken. To correct the problem, first use the COPY command to make copies of both files; then, delete the originals. Review each file for validity and edit as necessary.

File size error for file <filename>
The size of the file in a directory is different from its actual size. The size in the directory is automatically adjusted to indicate its actual size on the disk. (The amount of useful data may be less than the size shown because the last data block may not be used fully.)

XXXXXX bytes of disk space freed
Disk space shown as allocated was not actually allocated and has been freed.

NAME TYPE

COPY Internal

SYNTAX

COPY filespec [filespec]

FUNCTION

Copy the first filespec to the second.

COMMENTS

If the second filespec parameter is not given, the copy is on the default drive and has the same name as the original (first filespec parameter). If the first filespec is on the default drive and the second filespec is not given, the COPY is aborted. (Copying files to themselves is not allowed.) MS-DOS returns the error message:

```
File cannot be copied onto itself
0 File(s) copied
```

The second parameter may take three forms. If the second parameter is a drive designation (d:) only, the original file is copied with the same name to the designated drive. If the second parameter is a filename only, the original file is copied to a file with the name specified on the default drive. If the second parameter is a full filespec, the original file is copied to a file with the name specified on the designated drive.

The COPY command also allows file concatenation while copying. Concatenation is invoked by simply listing any number of files as parameters to COPY, separated by "+".

For example,

```
COPY  A.XYZ + B.COM+B:C.TXT BIGFILE.CRP
```

The above command concatenates the contents of A.XYZ, B.COM, and B:C.TXT and places them in the file on the default drive called BIGFILE.CRP.

The concatenation operation is normally carried out in text (or ASCII) mode, meaning a <CONTROL-Z> (1A hex) in the file is interpreted

as the end-of-file mark. To combine binary files, this interpretation of the end-of-file may be overridden with the /B switch, which forces the command to use the physical end-of-file as the end of file (that is, the file length seen in the DIR command).

For example,

```
COPY/B A.COM + B.COM
```

Also, in the above example, no resulting file name was given. In this case, COPY seeks to the end of A.COM and appends B.COM to it, leaving the result named A.COM.

ASCII and binary files may be arbitrarily combined by using /B on binary files and /A on ASCII files. A switch (/A or /B) takes effect on the file it is placed after and applies to all subsequent files until another switch is found.

A /A or /B switch on the destination file determines whether or not a <CONTROL-Z> is placed at the end of the file. (Source files read while /A is in effect have <CONTROL-Z> stripped off. If /A is in effect when the file is written, a single <CONTROL-Z> will be put back.) Thus, an additional <CONTROL-Z> would be appended with a command such as:

```
COPY  A.ASM/B B.ASM/A
```

This occurs because the /B on the first file prevents the <CONTROL-Z> from being stripped, and the /A on the second puts one on. The primary practical application may be the reverse, where a <CONTROL-Z> is stripped from the file.

For example:

```
COPY PROG.COM/B + ERRS.TXT/A NEWPROG.COM/B
```

It is assumed here that ERRS.TXT was generated by an editor, but is actually considered constant data (error messages) by the program it is being appended to. Since the result is a .COM file, a <CONTROL-Z> at the end is not needed.

Even when not concatenating files, the /A and /B switches are still processed. When not concatenating, the copy command defaults to binary copy. By using the /A switch, the result file may be truncated at the first end-of-file mark:

```
COPY A.TXT/A B.TXT
```

B.TXT may be shorter than A.TXT if A.TXT contained an embedded <CONTROL-Z>. B.TXT will have exactly one <CONTROL-Z>, the last character of the file.

Concatenation with ambiguous file names is allowed, and the COPY command normally "does what you want". To combine several files specified with an ambiguous name into one file, use a command like:

```
COPY  *.LST  COMBIN.PRN
```

All files matching *.LST are combined into one file called COMBIN.PRN. Another type of task is performing several individual concatenations:

```
COPY  *.LST + *.REF *.PRN
```

In this example, for each file found matching *.LST, that file is combined with the corresponding .REF file, with the result given the same name but with the extension .PRN. Thus, FILE1.LST will be combined with FILE1.REF to form FILE1.PRN, then XYZ.LST with XYZ.REF to form XYZ.PRN, and so on. The following COPY command combines all files matching *.LST, then all files matching *.REF, into one file call COMBIN.PRN:

```
COPY *.LST + *.REF COMBIN.PRN
```

It is easy to enter a concatenation COPY command where one of the source files is the same as the destination, yet this often cannot be detected. For example, the following command is an error if ALL.LST already exists:

```
COPY  *.LST  ALL.LST
```

This is not detected, however, until it is ALL.LST's turn to be appended. At this point it could already have been destroyed.

COPY handles this problem like this: as each input file is found, its name is compared with the destination. If they are the same, that one input file is skipped, and the message "Content of destination lost before copy" is printed. Further concatenation proceeds normally. This allows "summing" files, with a command like

```
COPY  ALL.LST + *.LST
```

This command appends all *.LST files, except ALL.LST itself, to ALL.LST. The error message is suppressed in this case, since this is produced by a true physical append to ALL.LST.

NAME **TYPE**

DATE Internal

SYNTAX

DATE [<mm>-<dd>-<yy>]

FUNCTION

Display and set the date.

COMMENTS

If entered without a parameter, DATE returns with the message:

```
Current date is <mm>-<dd>-<yy>
Enter new date:_
```

Press <RETURN> if you do not want to change the date shown.

Optionally, the date may be given as a parameter to the DATE command as in:

```
DATE 3-9-81
```

In this case, no message appears.

The new date must be entered using numerals only: letters are not permitted. The allowable parameters are:

```
<mm> = 1-12
<dd> = 1-31
<yy> = 80-99 or 1980-2099
```

The date, month, and year entries may be separated by hyphens (-) or slashes (/). MS-DOS is programmed to change months and years correctly, whether the month has 31, 30, 29, or 28 days. (Note that MS-DOS handles leap years, too.)

If the parameters or separators are not legal, MS-DOS returns the message:

```
Invalid date
Enter new date:_
```

DATE then waits for entry of a legal date.

NAME
DEL

TYPE
Internal

SYNTAX
DEL filespec

FUNCTION
Delete all the files with the filespec specified.

COMMENTS
If the filename is *.*, the prompt "Are you sure?" appears. If a "Y" or "y" is typed as a response, then all files are deleted as requested. ERASE is a synonym for this command.

NAME **TYPE**

DIR Internal

SYNTAX

DIR [filespec][/P][/W]

FUNCTION

List the files in a directory

COMMENTS

If no parameter is present (DIR), all directory entries on the default drive are listed. If only the drive specification is present (DIR d:), all entries on the disk in the specified drive are listed. If only filename is present (DIR filename) with no extension, then all files with the filename specified on the disk in the default drive are listed. If a full file specification is present (DIR d:filename.ext), all files with the filename specified on the disk in the drive specified are listed. In all cases, files are listed with their size in bytes and the time and date of their last modification.

The wild card characters question mark (?) and asterisk (*) may be used in the filename parameter. Refer to Section 2.2.2, "Filenames," for examples of the use of the wild card characters. Note that for the convenience of the user, the following invocations of the DIR command are equivalent:

COMMAND	EQUIVALENT
DIR	DIR *.*
DIR FILE	DIR FILE.*
DIR .EXT	DIR *.EXT
DIR .	DIR *.

Two switches may be given with DIR. The /P switch selects Page Mode. With /P, display of the directory pauses after the screen is filled. To resume display of output, type any key.

The /W switch selects Wide Display. With /W, only file names are displayed without other file information. Files are displayed five per line.

NAME TYPE

EXE2BIN External

SYNTAX

EXE2BIN filespec [d:][filename][.ext]

FUNCTION

Convert files from .EXE format to binary format

COMMENTS

The first parameter is the input file; if no extension is given, it defaults to .EXE. The second parameter is the output file. If no drive is given, the drive of the input file is used; if no filename is given, the filename of the input file is used; if no extension is given, .BIN is used.

The input file must be in valid .EXE format produced by the linker. The "resident", or actual code and data part of the file, must be less than 64K. There must be no STACK segment. Two kinds of conversion are possible depending on the specified initial CS:IP:

1. If CS:IP is not specified, a pure binary conversion is assumed. If segment fix-ups are necessary, the following prompt appears:

   Fix-up needed - base segment (hex):

   By typing a legal hexadecimal number and then <RETURN>, execution will continue.

2. If CS:IP is specified as 100H, then it is assumed the file is to be run as a .COM file ORGed at 100H, and the first 100H of the file is to be deleted. No segment fix-ups are allowed, as .COM files must be segment relocatable.

If CS:IP does not meet one of these criteria or meets the .COM file criterion, but has segment fix-ups, the following error message is displayed:

File cannot be converted

Note that to produce standard .COM files with the MACRO-86 assembler, one must both ORG the file at 100H and specify the first location as

the start address (this is done in the END statement).

For example:

```
        ORG     100H
START:
          .
          .
          .
        END     START
```

NAME TYPE

FORMAT External

SYNTAX

FORMAT d:[/S]

FUNCTION

Format the disk in the drive designated to accept MS-DOS files.

COMMENTS

Initialize the directory and file allocation tables. The reserved sectors are copied onto track 0, sector 1. (This occurs whether or not the /S switch is given.)

If the /S switch is present, FORMAT copies operating system files from the disk in the default drive to the newly formatted disk. The files copied are copied in the following order:

IO.SYS
MSDOS.SYS
COMMAND.COM

NAME TYPE

PAUSE Internal

SYNTAX

PAUSE [comment]

FUNCTION

Suspend execution of the batch file.

COMMENTS

During the execution of a batch file, you may need to change disks or to perform some other action between the execution of batch commands. The PAUSE command exists for just such purposes. PAUSE suspends execution until you type any key, except <CONTROL-C>.

When COMMAND encounters PAUSE, it prints:

Strike a key when ready . . .

Pressing any key except <CONTROL-C> resumes execution of the batch file. If you type <CONTROL-C>, another prompt is displayed:

Abort batch job (Y/N)?

If you type "Y" in response to this prompt, execution of the remainder of the batch command file is aborted and control returns to the operating system command level. Therefore, PAUSE can be used to break a batch file into pieces, allowing you to end the batch command file at an intermediate point.

The optional comment may be entered on the same line as PAUSE. You may also want to prompt the user of the batch file with some meaningful message when the batch file has paused. For example, you may want to change disks in one of the drives. An optional prompt message may be given in such cases. The comment prompt is displayed before the "Strike a key" message.

NAME
REM

TYPE
Internal

SYNTAX
REM [comment]

FUNCTION
Display remark entered on same line as REM when encountered during execution of batch file.

COMMENTS
The REM command has no other effect. The only delimiters for the comment are any one of the three legal delimiters to start the comment (blank space, tab, comma).

NAME
REN

TYPE
Internal

SYNTAX
REN filespec filename

FUNCTION
Change the name of the first parameter (filespec) to the second parameter (filename).

COMMENTS
The first parameter (filespec) must be given a drive designation if the file disk resides in a drive other than the currently logged (default) drive. Any drive designation for the second parameter (filename) is ignored. The file will remain on the disk where it currently resides.

The wildcard characters, question mark (?) and asterisk (*), may be used in either parameter. All files matching the first filespec are renamed. If wildcard characters appear in the second name, corresponding character positions are not changed.

For example, the following command changes the names of all files with the .LST extension to similar names with the .PRN extension:

```
REN *.LST *.PRN
```

Another example causes the file ABODE on drive B: to be renamed ADOBE:

```
REN B:ABODE ?D?B?
```

The file remains on drive B:.

An attempt to rename a file to a name already present in the directory will result in the error message "Duplicate file name or file not found."

Note that RENAME is a synonym for the REN command.

NAME
SYS

TYPE
External

SYNTAX
SYS d:

FUNCTION
Transfer the MS-DOS system files from the disk in the default drive to the drive specified by d:.

COMMENTS
SYS is normally used to update a system or to place the system on a formatted disk that contains no files. An entry for d: is required.

The target disk must either be completely blank or already have the hidden system files IO.SYS and MSDOS.SYS (see below). If this is not the case, SYS will report "No room for system on destination disk."

If IO.SYS and MSDOS.SYS are present, they must take up the same amount of space on the disk as the new system will need. If they do not, SYS will report "Incompatible system size."

The files transferred are copied in the following order:

IO.SYS
MSDOS.SYS

Note that COMMAND.COM is not transferred and that IO.SYS and MSDOS.SYS are both hidden files that do not appear when the DIR command is executed.

NAME

TIME

TYPE

Internal

SYNTAX

TIME [<hh>[:<mm>[:<ss>]]]

FUNCTION

Display and set the time.

COMMENTS

If the TIME command is entered without any parameters, then the following message is displayed:

```
Current time is <hh>:<mm>:<ss>.<cc>
Enter new time:_
```

Simply type <RETURN> if you do not want to change the time shown. Optionally, a new time may be given as a parameter to the TIME command as in:

```
TIME 8:20:00
```

The new time must be entered using numerals only: letters are not allowed. The allowable parameters are:

```
<hh> = 00-24
<mm> = 00-59
<ss> = 00-59
```

The hour, minute, and second entries must be separate by colons.

MS-DOS uses whatever time is entered as the new time as long as the parameters and separators are legal. If the parameters or separators are not legal, MS-DOS returns the message:

```
Invalid time
Enter new time:_
```

MS-DOS then waits for entry of a legal time.

NAME

TYPE

TYPE

Internal

SYNTAX

TYPE filespec

FUNCTION

Display the contents of the file on the console screen.

COMMENTS

Use this command to examine a file without modifying it. Use DIR to find the name of a file and EDLIN to alter the contents of a file. The only formatting performed by TYPE is that tabs are expanded to spaces consistent with tab stops every eighth column. Note that display of binary files causes control characters to be sent to your computer, including bells, formfeeds, and escape sequences.

CHAPTER 4

EDLIN

EDLIN is a text editor used to edit files that are divided into lines. Each line may be up to 255 characters, the last character of each being the end of line character, the carriage return. Line numbers are not actually present in saved text, but when a file is displayed, lines are numbered dynamically. When a file is created or edited, line numbers begin at 1 and are incremented by one through the end of the file. When new lines are inserted between existing lines, all line numbers following the inserted text are automatically incremented by the number of lines inserted. When lines are deleted between existing lines, all line numbers following the deleted text are decremented automatically by the number of lines deleted. Consequently, lines numbers always run from 1 through n (the last number).

4.1 INVOCATION

To invoke EDLIN, enter:

```
EDLIN <filespec>
```

If the file specified exists, EDLIN loads the file into memory. If the whole file is loaded, EDLIN returns the message "End of input file" and an asterisk (*) prompt. If the file is "larger than memory", then EDLIN fills 3/4 of available memory with the first part of the file and then returns the asterisk (*) prompt, but not the "End of input file" message. (This is just like the Append command with no parameter. See Section 4.3, "Interline Commands," for more information on Append.)

You may then edit the existing file. When you want to edit the part of a file that is not in memory, you must first write out to disk some of the file that is in memory, and then append lines into memory. (See the A and W commands.)

If the file specified does not exist, EDLIN creates the file and returns the message NEW FILE. and then displays the asterisk (*) prompt, indicating that the editing session may begin.

IMPORTANT

> When creating a new file, be sure to specify on which drive the file should be saved. The command to end the editing session and save the file does not allow parameters. Therefore, if the drive is not designated during EDLIN invocation, the file is saved on the default drive.

EDLIN commands belong to two types: intraline and interline. Intraline commands perform editing functions within a single line. The commands used to perform intraline editing are the control character functions and the special editing commands discussed in Chapter 2. The special editing functions are described in more detail in the following section than they were in Chapter 2. Note, however, that these are the same commands that are used at the MS-DOS command level. The only difference between them is that the EDLIN commands operate on the line currently being edited, rather than the MS-DOS command line.

4.2 INTRALINE COMMANDS

Intraline commands include the special editing functions and the control character functions: only the special editing functions are discussed here. See Section 2.3.1.2 for more information on the control character functions.

The special editing commands all may be assigned to the keys that make the best use of a specific terminal keyboard. Therefore, each command is identified by a functional name rather than by a specific key, and each is configurable to a particular keyboard key code. A code has been given to each command for ease of reference during the examples which demonstrate the function. (For an application on a specific terminal, the codes should be replaced by the names of the specific terminal keys.) Table 4.1 summarizes the commands, codes, and functions. Descriptions of the special editing functions follow the table.

Table 4.1 Special Editing Commands

Command	Code	Function
Copy One character	<C1>	Copy one character from the template to the new line
Copy up to character	<CM>	Copy all characters from the template to the new line up to the character specified
Copy Template	<CT>	Copy all remaining characters in the template to the new line
Skip One character	<S1>	Do not copy (skip over) a character in the template
Skip up to character	<SM>	Do not copy (skip over) the characters in the template up to the character specified
Quit Input	<QI>	Void the current input; leave the template unchanged
Insert mode	<INS>	Enter insert mode
Replace mode	<REP>	Exit insert mode (toggle from insert); this is the default
New Template	<NT>	Make the new line the new template

KEY

<C1>

FUNCTION

Copy one character from the template to the input buffer.

COMMENTS

Pressing the <C1> key copies one character from the template to the input buffer. When the <C1> key is pressed, one character is inserted in the buffer and insert mode is automatically turned off if it was on. Use the <C1> key to advance the cursor one column across the line.

EXAMPLE

Assume the screen shows:

```
1:*This is a sample file.
1:*_
```

At the beginning of the intraline edit, the cursor is positioned at the beginning of the line (indicated by the underline). Pressing the <C1> key copies the first character (T) to the second of the two lines displayed:

```
     1:*This is a sample file
<C1> 1:*T_
```

Each time the <C1> key is pressed, one more character appears:

```
<C1> 1:*Th_
<C1> 1:*Thi_
<C1> 1:*This_
```

KEY

<CM>

FUNCTION

Copy multiple characters up to a given character

COMMENTS

Pressing the <CM> key copies all characters up to a given character from the template to the input buffer. The given character is the next character typed and is not copied or shown on the screen. Pressing the <CM> key causes the cursor to move to the single character that is this command's only parameter. If the template does not contain the specified character, nothing is copied. Pressing <CM> also automatically turns off insert mode if it is on.

EXAMPLE

Assume the screen shows:

```
1:*This is a sample file.
1:*_
```

At the beginning of the intraline edit, the cursor is positioned at the beginning of the line (indicated by the underline). Pressing the <CM> key copies all characters up to the character pressed immediately after the <CM> key.

```
          1:*This is a sample file
<CM>p     1:*This is a sam_
```

KEY

<CT>

FUNCTION

Copy template to input buffer

COMMENTS

Pressing <CT> copies all remaining characters from the template to the input buffer. Regardless of the cursor position at the time the <CT> key is pressed, the rest of the line appears, and the cursor is positioned after the last character on the line.

EXAMPLE:

Assume the screen shows:

```
1:*This is a sample file.
1:*_
```

At the beginning of the intraline edit, the cursor is positioned at the beginning of the line (indicated by the underline). Pressing the <CT> key copies all characters from the template (shown in the upper line displayed) to the line with the cursor (the lower line displayed):

```
     1:*This is a sample file
<CT> 1:*This is a sample file._
```

Also, insert mode is automatically turned off if it was on.

KEY

<S1>

FUNCTION

Skip over one character in the template

COMMENTS

Pressing the <S1> key skips over one character in the template. Each time you press the <S1> key, one character is deleted (not copied) from the template. The action of the <S1> key is similar to the <C1> key, except that <S1> skips a character in the template rather than copies it to the input buffer.

EXAMPLE:

Assume the screen shows:

```
1:*This is a sample file.
1:*_
```

At the beginning of the intraline edit, the cursor is positioned at the beginning of the line (indicated by the underline). Pressing the <S1> key skips over the first character ("T").

```
     1:*This is a sample file
<S1> 1:*_
```

The cursor position does not move, only the template is affected. To see how much of the line has been (skipped over), press the <CT> key, which moves the cursor beyond the last character of the line.

```
     1:*This is a sample file.
<S1> 1:*_
<CT> 1:*his is a sample file._
```

KEY

<SM>

FUNCTION

Skip multiple characters in the template

COMMENTS

Pressing the <SM> key skips over all characters up to a given character in the template. The given character is the next character typed, and is not copied and not shown on the screen. If the template does not contain the specified character, nothing is skipped over. The action of the <SM> key is similar to the <CM> key, except that <SM> skips over characters in the template rather than copies them to the input buffer.

EXAMPLE

Assume the screen shows:

```
1:*This is a sample file.
1:*_
```

At the beginning of the intraline edit, the cursor is positioned at the beginning of the line (indicated by the underline). Pressing the <SM> key skips over (deletes) all the characters in the template up to the character pressed after the <SM> key:

```
        1:*This is a sample file
<SM>p 1:*_
```

The cursor position does not move. To see how much of the line has been skipped over, press the <CT> key to copy the template. This moves the cursor beyond the last character of the line:

```
        1:*This is a sample file:
<SM>p 1:*_
<CT>  1:*ple file._
```

KEY

<QI>

FUNCTION

Quit input and flush the input buffer.

COMMENTS

Pressing the <QI> key flushes the input buffer, but it leaves the template unchanged. <QI> also prints a back slash (\), carriage return, and line feed, and turns insert mode off if it was on. The cursor is positioned at the beginning of the line. Pressing the <CT> key copies the template to the input buffer just as the line was before <QI> was pressed.

EXAMPLE

Assume the screen shows:

```
1:*This is a sample file.
1:*_
```

At the beginning of the intraline edit, the cursor is positioned at the beginning of the line (indicated by the underline). Assume you want to replace the line by typing:

```
                1:*This is a sample file.
Sample File 1:*Sample File_
```

Now, to reedit the line, press <QI>:

```
     1:*This is a sample file.
<QI> 1:*Sample File\
     1:_
```

<RETURN> can now be pressed to keep the original line or to perform any other intraline editing functions. If <CT> is pressed, the original template is copied to the input buffer:

```
<CT> 1: This is a sample file._
```

KEY

<INS>

FUNCTION

Enter insert mode

COMMENTS

Pressing the <INS> key causes entry into insert mode. The current position in the template is not changed. The cursor does move as each character is inserted. However, when you have finished inserting characters, the cursor is positioned at the same character as it was before the insertion began. Thus, characters are inserted before the character the cursor points to.

EXAMPLE

Assume the screen shows: .

```
1:*This is a sample file.
1:*_
```

At the beginning of the intraline edit, the cursor is positioned at the beginning of the line (indicated by the underline). Assume you press the <CM> and "p" keys:

```
          1:*This is a sample file
<CM>p 1:*This is a sam_
```

Now press the INS key and insert the three characters "s", "o", and "n".

```
               1:*This is a sample file.
<CM>p          1:*This is a sam_
<INS>son 1:*This is a samson_
```

If you now press the <CT> key, the rest of the template is copied to the line:

```
          1:*This is a samson_
<CT> 1:*This is a samsonple file._
```

If you were to press the <RETURN> key, instead, the remainder of the template would be truncated, and the input buffer ended at the end of the insert:

```
<INS>son<RETURN> 1:*This is a samson
```

KEY

<REP>

FUNCTION

Enter Replace mode.

COMMENTS

Pressing the <REP> key causes exit from insert mode and entry into replace mode. All characters entered overstrike and replace characters in the template. (Replace mode is the default.) When you start to edit a line, this mode is in effect. Each character typed replaces a character in the template. If the <RETURN> key is pressed, the remainder of the template is truncated.

EXAMPLE

Assume the screen shows:

```
1:*This is a sample file.
1:*_
```

At the beginning of the intraline edit, the cursor is positioned at the beginning of the line (indicated by the underline). Assume you then press <CM>p, <INS>son, <REP>ite, and then <CT>:

```
                1:*This is a sample file.
<CM>p           1:*This is a sam_
<INS>son        1:*This is a samson_
<REP>ite        1:*This is a samsonite_
<CT>            1:*This is a samsonite file._
```

If you type in characters that extend beyond the length of the template, the remaining characters in the template are automatically appended when you type <CT>.

KEY

<NT>

FUNCTION

Create new template

COMMENTS

Pressing the <NT> Key copies the current contents of the input buffer to the template. The contents of the old template are then destroyed. Pressing <NT> outputs an at sign character (@), a carriage return, and a line feed. The input buffer is also emptied and insert mode is turned off.

NOTE

<NT> performs the same functions as the <QI> key, except that the template is changed and an at sign character (@) is printed instead of a backslash (-).

EXAMPLE

Assume the screen shows:

```
1:*This is a sample file.
1:*_
```

At the beginning of the intraline edit, the cursor is positioned at the beginning of the line (indicated by the underline). Assume that you enter <CM>p, <INS>son, <REP>ite, and then <CT>:

```
                1:*This is a sample file.
<CM>p           1:*This is a sam_
<INS>son        1:*This is a samson_
<REP>ite        1:*This is a samsonite_
<CT>            1:*This is a samsonite file._
```

At this point, assume that you want this line as the new template, so you press the <NT> key:

```
<NT> 1:*This is a samsonite file.@
       _
```

Additional editing can now be done using the above new template.

4.3 INTERLINE COMMANDS

Interline commands perform editing functions on whole lines at a time. The interline commands are summarized in the following list and are described in detail with examples following the description of command parameters.

Table 4.1 Interline Commands

Command	Purpose
<line>	Edit Line
A	Append Lines
D	Delete Lines
E	End Editing
I	Insert Text
L	List Text
Q	Quit Editing
R	Replace Text
S	Search Text
W	Write Lines

4.3.1 Parameters

Each interline command accepts some optional parameters. The following list of parameters indicates their form. The effect of a parameter depends on the command it is used with.

PARAMETER DEFINITION

<line> <line> indicates a line number to be entered by the user. Line numbers must be separated from other line numbers, other parameters, and the command. Use a comma or space to separate.

<line> may be specified one of three ways:

Number any integer less than 65534. If a number larger than the largest existing line number is specified, then <line> indicates the line after the last line number.

Period (.) If a period is specified for <line>, then <line> indicates the current line number. The current line is the last line edited, and not necessarily the last line displayed. The current line is marked on your screen by an asterisk (*) between the line number and the first character.

Pound (#) The pound sign indicates the line after the last line number. Specifying # for <line> has the same effect as specifying a number larger than the last line number.

<RETURN> A carriage return entered without any of the <line> specifiers listed above directs EDLIN to use a default value appropriate to the command.

? The question mark parameter directs EDLIN to ask the user if the correct string has been found. The question mark is used only with the Replace and Search commands. Before continuing, EDLIN waits for either a "Y" or <RETURN> for a yes response, or for any other key for a no response.

<string> <string> represents text to be found, to be replaced, or to replace other text. The <string> parameter is used only with the Search and Replace commands. Each <string> must be terminated by a <CONTROL-Z> or a <RETURN> (see the Replace command for details). No spaces should be left between strings or between a string and its command letter, unless you want spaces as part of a string.

NAME

Edit

SYNTAX

[<line>]

FUNCTION

Edit line

COMMENTS

When a line number is entered, EDLIN displays the line number and text, then, on the line below, reprints the line number. The line is then ready for editing. You may use any of the available intraline commands to edit the line. The existing text of the line serves as the template until the <RETURN> key is pressed.

If no line number is entered (that is, only the <RETURN> key is pressed), the line after the current line, marked with an asterisk (*), is edited. If no changes of the current line are needed and the cursor position is at the beginning or end of the line, press the <RETURN> key to accept the line as is.

WARNING

If the <RETURN> key is pressed while the cursor is in the middle of the line, the remainder of the line is truncated.

EXAMPLE

Assume the following file exists and is ready to edit:

```
1: This is a sample file.
2: used to demonstrate
3: the editing of line
4:* four.
```

To edit line 4, enter:

```
4
```

The contents of the line are displayed along with a cursor below the line:

```
4:* four.
4:*_
```

Now type:

```
<INS>number  4: number_
<CT><RETURN> 4: number four.
             5:*_
```

NAME

Append

SYNTAX

[<n>]A

FUNCTION

Append lines from input file to editing buffer

COMMENTS

Use this command for extremely large files that will not fit into memory all at one time. By writing out part of the editing buffer to the output file with the Write command, room is made for lines to be appended with the Append command. If A is typed without a parameter, lines are appended to the part of the file currently in memory until available memory is 3/4 full or until there are no more lines to append.

Use the W command to write out lines to the output file. If the parameter <n> is given, then <n> lines are appended to that part of the file that currently is in memory. If <n> is not given, then as much of the input file as possible is read into the editing buffer until the editing buffer is three quarters full.

NAME

Delete

SYNTAX

[<line>][,<line>] D

FUNCTION

Delete the specified lines and all lines in between

COMMENTS

If the first <line> is omitted, the first <line> defaults to the current line (the line with the asterisk next to the line number). If the second <line> is omitted, then just the first <line> is deleted. When lines have been deleted, the line immediately after the deleted section becomes the current line and has the same line number as the first <line> had before the deletion occurred.

EXAMPLE

Assume the following file exists and is ready to edit:

```
   1: This is a sample file.
   2: Use: to demonstrate dynamic line numbers
   3: See what happens when you
   4: Delete and Insert
    .
    .
    .
  25: (The D and I commands)
  26: (Use <CONTROL-C> to exit insert mode)
  27:*Line numbers
```

To delete multiple lines, enter <line>,<line> D:

```
5,24 D
```

The result is:

```
1: This is a sample file.
2: Use: to demonstrate dynamic line numbers
3: See what happens when you
4: Delete and Insert
5*(The D and I commands)
6: (Use <CONTROL-C> to exit insert mode)
7: Line numbers
```

To delete a single line, enter:

```
6 D
```

The result is:

```
1: This is a sample file.
2: Use: to demonstrate dynamic line numbers
3: See what happens when you
4: Delete and Insert
5: (The D and I commands)
6:*Line numbers
```

Next, delete a range of lines from the following file:

```
1: This is a sample file.
2: Use: to demonstrate dynamic line numbers
3:*See what happens when you
4: Delete and Insert
5: (The D and I commands)
6: (Use <CONTROL-C> to exit insert mode)
7: Line numbers
```

To delete beginning with the current line, enter:

```
,6 D
```

The result is:

```
1: This is a sample file.
2: Use: to demonstrate dynamic line numbers
3:*Line numbers
```

NAME

End

SYNTAX

E

FUNCTION

End the editing session

COMMENTS

Save the edited file on disk, rename the original input file "filename.BAK", and then exit EDLIN to the MS-DOS command level. If the file was created during the editing session, no .BAK file is created.

The E command takes no parameters. Therefore, you cannot tell EDLIN on which drive to save the file. The drive must be selected when the editing session is invoked. If the drive is not designated when EDLIN was invoked, the file is saved on the disk in the default drive. (It will still be possible to COPY the file to a different drive. However, this is done automatically if the drive is designated during invocation.)

You must be sure that the disk contains enough free space for the entire file to be written. If the disk does not contain enough free space, the write is aborted and the edited file lost, although part of the file may be written out.

EXAMPLE

The only possible command is:

E<RETURN>

After execution of the E command, control is returned to COMMAND.COM and the MS-DOS prompt is displayed.

NAME

Insert

SYNTAX

[<line>] I

FUNCTION

Insert line(s) of text immediately before the specified <line>.

COMMENTS

If you are creating a new file, the I command must be given before text can be inserted. In this case, the insert begins with line number 1.

EDLIN remains in insert mode until a <CONTROL-Z> or a <CONTROL-C> is entered. Successive line numbers appear automatically each time <RETURN> is pressed. When the insert is finished and insert mode has been exited, the <line>, which now immediately follows the inserted lines, becomes the current line. All line numbers following the inserted section are incremented by the number of lines inserted.

If <line> is not specified, the default is the current line number (the lines are inserted immediately before the current line). If <line> is an integer larger than the last line number, or if # is specified as <line>, the inserted lines are appended to the end of the file. In this case, the last line inserted becomes the current line. (This is the same as when the file is being created.)

EXAMPLE

Assume the following file exists and is ready to edit:

```
1: This is a sample file.
2: Use: to demonstrate dynamic line numbers
3: See what happens when you
4: Delete and Insert
5: (The D and I commands)
6: (Use <CONTROL-C> to exit insert mode)
7:*Line numbers
```

To insert text before a specific line (not the current line), enter <line> I:

```
4 I
```

The result is:

```
4:_
```

Now, enter the new text for lines 4 and 5:

```
4: fool around with
5: those very useful commands that
```

Then to end the insertion, type:

```
6: <CONTROL-Z>
```

Now type L to list the file; the result is:

```
1: This is a sample file.
2: Use: to demonstrate dynamic line numbers
3: See what happens when you
4: fool around with
5: those very useful commands that
6:*Delete and Insert
7: (The D and I commands)
8: (Use <CONTROL-C> to exit insert mode)
9: Line numbers
```

To insert lines immediately before the current line, enter:

```
I
```

The result is:

```
6: _
```

Now, insert the following text terminated with a <CONTROL-Z>:

```
6: perform the two major editing functions,
```

Now to List the file and see the result, type:

```
L
```

The result is:

```
 1: This is a sample file.
 2: Use: to demonstrate dynamic line numbers
 3: See what happens when you
 4: fool around with
 5: those very useful commands that
 6: perform the two major editing functions,
 7:*Delete and Insert
 8: (The D and I commands)
 9: (Use <CONTROL-C> to exit insert mode)
10: Line numbers
```

To append new lines to the end of the file, enter:

```
11 I
```

This produces the following:

```
11: _
```

Now, enter the following new lines:

```
11: The insert command can place new lines
12: anywhere in the file; there's no space problems.
13: because the line numbers are dynamic;
14: They'll slide all the way to 65533.
```

End insertion by typing <CONTROL-C>. The new lines will appear at the end of all previous lines in the file. Now enter the list command:

```
L
```

The result is:

```
 1: This is a sample file.
 2: Use: to demonstrate dynamic line numbers
 3: See what happens when you
 4: fool around with
 5: those very useful commands that
 6: perform the two major editing functions,
 7: Delete and Insert
 8: (The D and I commands)
 9: (Use <CONTROL-C> to exit insert mode)
10: Line numbers
11: The insert command can place new lines
12: anywhere in the file; there's no space problems.
13: because the line numbers are dynamic;
14: They'll slide all the way to 65533.
```

NAME

List

SYNTAX

[<line>][,<line>] L

FUNCTION

List the specified range of lines, including the two lines specified.

COMMENTS

If the first <line> is omitted, the first <line> defaults to the current line. If the second <line> is omitted, 23 lines are listed; the eleven lines before <line>, the <line>, and the eleven lines after <line>. The current line remains unchanged. If the current line is one of the lines listed, it contains an asterisk between the line number and the first character.

EXAMPLE

Assume the following file exists and is ready to edit:

```
 1: This is a sample file.
 2: Use: to demonstrate dynamic line numbers
 3: See what happens when you
 4: Delete and Insert
 5: (The D and I commands)
  .
  .
  .
15:*The current line contains an asterisk.
  .
  .
  .
26: (Use <CONTROL-C> to exit insert mode)
27: Line numbers
```

To list a range of lines without reference to the current line, enter <line>,<line> L:

```
2,5 L
```

The result is:

```
2: Use: to demonstrate dynamic line numbers
3: See what happens when you
4: Delete and Insert
5: (The D and I commands)
```

To list a range of lines beginning with the current line, enter ,<line> L:

```
,26 L
```

The result is:

```
15:*The current line contains an asterisk.
   .
   .
   .
26: (Use <CONTROL-C> to exit insert mode)
```

To list a range of 23 lines around a specified line, enter <line>, L:

```
13, L
```

The result is:

```
13: The specified line is listed first in the range.
14: The current line remains unchanged by the L command.
15:*The current line contains an asterisk.
   .
   .
   .
35: <CONTROL-C> exits interline insert command mode.
```

To list a range of 23 lines centered around the current line, enter only L:

```
L
```

The result is:

```
 2: Use: to demonstrate dynamic line numbers
 3: See what happens when you
 4: Delete and Insert
 5: (The D and I commands)
   .
   .
   .
13: The current line is listed in the middle of the range.
14: The current line remains unchanged by the L command.
15:*The current line contains an asterisk.
   .
   .
   .
24: <CONTROL-C> exits interline insert command mode.
```

NAME

Quit

SYNTAX

Q

FUNCTION

Quit the editing session, do not save any editing changes, and exit to the MS-DOS operating system.

COMMENTS

No .BAK file is created. The Q command takes no parameters. It is simply a fast means of exiting an editing session. As soon as the Q command is given, EDLIN displays the message:

```
Abort edit (Y/N)?_
```

Press "Y" to quit the editing session;press "N" (or any other key except <CONTROL-C>) if you decide to continue the editing session.

EXAMPLE

Assume the following file exists and is ready to edit:

```
1: This is a sample file.
2: Use: to demonstrate dynamic line numbers
3: Compare the before and after
4: See what happens when you
5: Delete and Insert
6: Line numbers
```

Now, to delete line 3, enter:

```
3 D
```

To list the file, enter "L":

```
1: This is a sample file.
2: Use: to demonstrate dynamic line numbers
3: See what happens when you
4: Delete and Insert
5: Line numbers
```

Now, to keep the changes and to quit the editing session, enter:

```
Q
```

The result is:

```
Abort edit (Y/N)?_
```

Enter "Y" to exit to the operating system command level:

```
Abort edit (Y/N)?Y
A:_
```

NAME

Replace

SYNTAX

[<line>][,<line>] [?] R<stringl><CONTROL-Z><string2>

FUNCTION

Replace all occurrences of <stringl> in the specified range with <string2>.

COMMENTS

As each occurrence of <stringl> is found, it is replaced by <string2>. Each line in which a replacement occurs is displayed. If a line contains two or more replacemments of <stringl> with <string2>, then the line is displayed once for each occurrence. When all occurrences of <stringl> in the specified range are replaced by <string2>, the R command terminates and the asterisk prompt reappears.

If a second string is to be given as a replacement, then <stringl> must be terminated with a <CONTROL-Z>. If the <string2> is to be omitted, then <stringl> may be terminated with either a combination <CONTROL-Z><RETURN>, or simply a <RETURN>. <String2> must also be terminated with a <CONTROL-Z><RETURN> combination or with a simple <RETURN>. If <stringl> is omitted, then the replacement is terminated immediately. If <string2> is omitted, then <stringl> is deleted from all lines in the range. If the first <line> is omitted in the range argument (as in ,<line>) then the first <line> defaults to the line after the current line. If the second <line> is omitted (as in <line> or <line>,) the second <line> defaults to #. Therefore, this is the same as <line>,#. Remember that # indicates the line after the last line of the file.

If the question mark (?) parameter is given, the Replace command stops at each line with a string that matches <stringl>, displays the line with <string2> in place, and then displays the prompt "O.K.?". If the user presses "Y" or the <RETURN> key, then <string2> replaces <stringl>, and the next occurrence of <stringl> is found. Again, the "O.K.?" prompt is displayed. This process continues until the end of the range or until the end of the file. After the last occurrence of <stringl> is

found, EDLIN returns the asterisk prompt.

If you press any key besides "Y" or <RETURN> after the "O.K.?" prompt, the <string1> is left as it was in the line, and the Replace goes to the next occurrence of <string1>. If <string1> occurs more than once in a line, each occurrence of <string1> is replaced individually, and the "O.K.?" prompt is displayed after each replacement. In this way, only the desired <string1> is replaced, and you prevent replacement of embedded strings.

EXAMPLE

Assume the following file exists and is ready for editing:

```
 1: This is a sample file.
 2: Use: to demonstrate dynamic line numbers
 3: See what happens when you
 4: fool around with
 5: those very useful commands that
 6: perform the two major editing functions,
 7: Delete and Insert
 8: (The D and I commands)
 9: (Use <CONTROL-C> to exit insert mode)
10: Line numbers
11: The insert command can place new lines
12: anywhere in the file; there's no space problems.
13: because the line numbers are dynamic;
14: They'll slide all the way to 65533.
```

To replace all occurrences of <string1> with <string2> in a specified range, enter:

```
2,12 Rand<CONTROL-Z>or<RETURN>
```

The result is:

```
 5: those very useful commors that
 7: Delete or Insert
 8: (The D or I commands)
 8: (The D or I commors)
11: The insert commor can place new lines
```

Note that in the above replacement, some unwanted substitutions have occurred. To avoid these, and confirm each replacement, the same original file can be used:

```
    .
    .
 5: those very useful commands that
    .
 7: Delete and Insert
 8: (The D and I commands)
    .
11: The insert command can place new lines
    .
    .
```

only with a slightly different command. This time, to replace only certain occurrences of the first <string> with the second <string>, enter:

```
2? Rand<CONTROL-Z>or<RETURN>
```

The result is:

```
 5: those very useful commors that
O.K.? N
 7: Delete or Insert
O.K.? Y
 8: (The D or I commands)
O.K.? Y
 8: (The D or I commors)
O.K.? N
11: The insert commor can place new lines
O.K.? N
*_
```

Now, enter the List command (L) to see the result of all these changes:

```
    .
    .
 5: those very useful commands that
    .
 7: Delete or Insert
 8: (The D or I commands)
    .
11: The insert command can place new lines
    .
    .
```

NAME

Search

SYNTAX

[<line>][,<line>] [?] S<string>

FUNCTION

Search the specified range of lines for the specified string.

COMMENTS

The <string> must be terminated with a <RETURN>. The first line that matches <string> is displayed and becomes the current line. The Search command terminates when a match is found. If no line contains a match for <string>, the message "Not found" is displayed.

If the optional parameter, question mark (?), is included in the command, EDLIN displays the first line with a matching string; it then prompts the user with the message "O.K.?". If the user presses either the "Y" or <RETURN> key, the line becomes the current line and the search terminates. If the user presses any other key, the search continues until another match is found, or until all lines have been searched (then the "Not found" message is displayed).

If the first <line> is omitted (as in ,<line> S<string>), the first <line> defaults to the line after the current line. If the second <line> is omitted (as in <line> S<string> or <line>, S<string>), the second <line> defaults to #, which is the same as <line>,# S<string>. If <string> is omitted, no search is made and the command terminates immediately.

EXAMPLE

Assume the following file exists and is ready for editing:

```
1: This is a sample file.
2: Use: to demonstrate dynamic line numbers
3: See what happens when you
4: fool around with
5: those very useful commands that
6: perform the two major editing functions,
7: Delete and Insert
8: (The D and I commands)
```

```
 9: (Use <CONTROL-C> to exit insert mode)
10: Line numbers
11: The insert command can place new lines
12: anywhere in the file; there's no space problems.
13: because the line numbers are dynamic;
14:*They'll slide all the way to 65533.
```

To search for the first occurrence of a string, enter:

```
2,12 Sand<RETURN>
```

The result is:

```
5: those very useful commands that
```

To get the "and" in line 7, modify the search command by entering:

```
<S1><CT>,12 Sand<RETURN>
```

The search then continues from the line after the current line (line 5), since no first line is given. The result is:

```
7: Delete and Insert
```

To Search through several occurrences of a string until the correct string is found, enter:

```
1, ? Sand
```

The result is:

```
5: those very useful commands that
O.K.?_
```

If you press any key except "Y" or <RETURN>, the search continues, so type "N" here:

```
O.K.? N
```

Continue:

```
7 Delete and Insert
O.K.?_
```

Now press press "Y" to terminate the search:

```
O.K.? Y
*_
```

NAME

Write

SYNTAX

[<n>]W

FUNCTION

Write lines from the editing buffer to the output file

COMMENTS

The Write command is used when editing files that are larger than available memory. By executing the Write, lines are written out to the output file and room is made in the input buffer for more lines to be appended from the input file. If W is typed with no <n> parameter, then lines are written until memory is 1/4 full.

If the <n> parameter is given, then <n> lines are written out. Note that lines are written out beginning with the start of the file; subsequent lines in the editing buffer are renumbered beginning with one. A later Append command will append lines to any remaining lines in the editing buffer.

4.4 ERROR MESSAGES

EDLIN error messages occur either when you try to invoke EDLIN or during the actual editing session.

4.4.1 Errors When Invoking EDLIN

Cannot edit .BAK file--rename file

Cause: The user attempted to edit a file with the filename extension .BAK. .BAK files cannot be edited because the extension is reserved for backup copies.

Cure: If the user needs the .BAK file for editing purposes, the user must either RENAME the file with a different extension or COPY the .BAK file but with a different filename extension.

No room in directory for file

Cause: When the user attempted to create a new file, either the file directory was full or the user specified an illegal disk drive or an illegal filename.

Cure: Check the EDLIN invocation command line for illegal filename and illegal disk drive entries. If the command is no longer on the screen and if the user has not yet entered a new command, the EDLIN invocation command can be recovered by pressing the <CT> key.

If the invocation command line contains no illegal entries, run the CHKDSK program for the specified disk drive. If the status report shows the disk directory full, remove the disk and insert and format a new disk. If the CHKDSK status report shows the disk directory is not full, check the EDLIN invocation command for an illegal filename or illegal disk drive designation.

4.5 ERRORS WHILE EDITING

Entry Error

Cause: The last command entered contained a syntax error.

Cure: Reenter the command with the correct syntax.

Line too long

Cause: During Replace command mode, the string given as the replacement causes the line to expand beyond the limit of 254 characters. EDLIN aborts the Replace command.

Cure: Divide the long line into two lines, then retry the Replace command.

Disk Full--file write not completed

Cause: The user gave the End command, but the disk did not contain enough free space for the whole file. EDLIN aborts the E command and returns the user to the operating system. Some of the file may have been written to the disk.

Cure: Only a portion (at most) of the file will have been saved. The user should probably delete whatever file was saved and restart the editing session. None of the file not written out will be available after this error. Always be sure that the disk has sufficient free space for the file to be written, before you begin your editing session.

4.5 ERRORS WHILE EDITING

"ntry Error

Cause: The last command entered contained a syntax error.

Cure: Reenter the command with the correct syntax.

Line too long

Cause: During Replace command mode, the string given as the replacement causes the line to expand beyond the limit of 254 characters. EDLIN aborts the Replace command.

Cure: Divide the long line into two lines, then retry the Replace command.

Disk Full--file write not completed

Cause: The user gave the End command, but the disk did not contain enough free space for the whole file. EDLIN aborts the E command and returns the user to the operating system. Some of the file may have been written to the disk.

Cure: Only a portion (at most) of the file will have been saved. The user should probably delete whatever file was saved and restart the editing session. None of the file not written out will be available after this error. Always be sure that the disk has sufficient free space for the file to be written, before you begin your editing session.

CHAPTER 5

DEBUG

DEBUG is a debugging program used to provide a controlled testing environment for binary and executable object files. Note that EDLIN is used to alter source files; DEBUG is EDLIN's counterpart for binary files. DEBUG eliminates the need to reassemble a program to see if a problem has been fixed by a minor change. It allows you to alter the contents of a file or the contents of a CPU register, and then to immediately reexecute a program to check of the validity of the changes.

All DEBUG commands may be aborted at any time by pressing <CONTROL-C>. <CONTROL-S> suspends the display, so that the user can read it before the output scrolls away. Entering any key other than <CONTROL-C> or <CONTROL-S> restarts the display. All of these commands are consistent with the control character functions available at the MS-DOS command level.

5.1 INVOCATION

To invoke DEBUG, enter:

```
DEBUG [<filespec> [<arglist>] ]
```

For example, if a <filespec> is specified, then the following is a typical invocation:

```
DEBUG FILE.EXE
```

DEBUG then loads FILE.EXE into memory starting at 100 hexadecimal in the lowest available segment. The BX:CX registers are loaded with the number of bytes placed into memory. The DEBUG prompt is a right angle bracket (>).

An <arglist> may be specified if <filespec> is present. These are filename parameters and switches that are to be passed to the program <filespec>. Thus, when <filespec> is loaded into memory it is loaded as if it had been invoked with the command:

```
<filespec> <arglist>
```

Here, <filespec> is the file to be debugged, and the <arglist> is the rest of the command line that is used when <filespec> is invoked and loaded into memory.

If no <filespec> is specified, then DEBUG is invoked as follows:

```
DEBUG
```

DEBUG then returns with the prompt, signaling that it is ready to accept user commands. Since no filename has been specified, current memory, disk sectors, or disk files can be worked on by invoking later commands.

5.2 COMMANDS

Each DEBUG command consists of a single letter followed by one or more parameters. Additionally, the control character and the special editing functions described in Chapter 2, all apply inside DEBUG.

If a syntax error occurs in a DEBUG command, DEBUG reprints the command line and indicates the error with an up-arrow (^) and the word error.

For example:

```
dcs:100 cs:110
    ^ error
```

All commands and parameters may be entered in either upper or lower case. Any combination of upper and lower case may be used in commands.

The DEBUG commands are summarized in Table 5.1 and are described in detail with examples following the description of command parameters.

Table 5.1 DEBUG Commands

DEBUG Command	Function
C<range> <address>	Compare
D[<range>]	Dump
E<address> [<list>]	Enter
F<range> <list>	Fill
G[=<address> [<address>...]]	Go
H<address> <address>	Hex
I<value>	Input
L[<address> [<drive><record><record>]]	Load
M<range> <address>	Move
N<filespec>	Name
O<value> <byte>	Output
Q	Quit
R[<register-name>]	Register
S<range> <list>	Search
T[=<address>][<value>]	Trace
U[<range>]	Unassemble
W[<address> [<drive><record><record>]]	Write

5.3 PARAMETERS

As the summary above shows, all DEBUG commands accept parameters, except the Quit command. Parameters may be separated by delimiters (spaces or commas), but a delimiter is required only between two consecutive hexadecimal values. Thus, the following commands are equivalent:

```
dcs:100 110
d cs:100 110
d,cs:100,110
```

Also, entries may be made in any combination upper or lower case.

PARAMETER	DEFINITION
<drive>	A one digit hexadecimal value to indicate which drive a file will be loaded from or written to. The valid values are 0-3. These values designate the drives as follows: 0=A:, 1=B:, 2=C:, 3=D:.
<byte>	A two digit hexadecimal value to be placed in or read from an address or register.
<record>	A 1 to 3 digit hexadecimal value used to indicate the logical record number on the disk and the number of disk sectors to be written or loaded. Logical records correspond to sectors. However their numbering differs since they represent the entire disk space.
<value>	A hexadecimal value up to four digits used to specify a port number or the number of times a command should repeat its functions.
<address>	A two part designation consisting of either an alphabetic segment register designation or a four digit segment address plus an offset value. The segment designation or segment address may be omitted, in which case the default segment is used. DS is the default segment for all commands except G, L, T, U, and W, for which the default segment is CS. All numeric values are hexadecimal. For example: CS:0100 04BA:0100 The colon is required between a segment designation (whether numeric or alphabetic) and an offset.

<range> Two <address>s: e.g., <address> <address>; or one <address>, an L, and a <value>: e.g., <address> L <value> where <value> is the number of lines the command should operate on; or simply <address>, and L80 is assumed. The last form can not be used if another hex value follows the <range>, since the hex value would be interpreted as the second <address> of the <range>.

Examples:

```
CS:100 110
CS:100 L 10
CS:100
```

The following is illegal:

```
CS:100 CS:110
       ^ error
```

The limit for <range> is 10000 hex. To specify a <value> of 10000 hex within four digits, enter 0000 (or 0).

<list> A series of <byte> values or of <string>s.line. <list> must be the last parameter on the command line.

Example:

```
fcs:100 42 45 52 54 41
```

<string> Any number of characters enclosed in quote marks. Quote marks may be either single (') or double("). Within <string>s, the opposite set of quote marks may be used freely as literals. If the delimiter quote marks must appear within a <string>, the quote marks must be doubled. For example, the following strings are legal:

```
'This is a "string" is okay.'
'This is a ''string'' is okay.'
```

However, this string is illegal:

```
'This is a 'string' is not.'
```

Similarly, these strings are legal:

```
"This is a 'string' is okay."
"This is a ""string"" is okay."
```

However, this string is illegal:

```
"This is a "string" is not."
```

Note that the double quotations are not necessary in the following strings:

```
"This is a ''string'' is not necessary."
'This is a ""string"" is not necessary.'
```

The ASCII values of the characters in the string are used as a <list> of byte values.

NAME

Compare

SYNTAX

C<range> <address>

FUNCTION

Compare the portion of memory specified by <range> to a portion of the same size beginning at <address>.

COMMENTS

If the two areas of memory are identical, there is no display and DEBUG returns with the MS-DOS prompt. If there are differences, they are displayed as:

<address1> <byte1> <byte2> <address2>

EXAMPLE

The following commands have the same effect:

C100,1FF 300

or

C100L100 300

Each command compares the block of memory from 100 to 1FFH with the block of memory from 300 to 3FFH.

NAME

Dump

SYNTAX

D[<range>]

FUNCTION

Display the contents of the specified region of memory.

COMMENTS

If a range of addresses is specified, the contents of the range are displayed. If the D command is entered without parameters, 128 bytes are displayed at the first address (DS:100) after that displayed by the previous Dump command.

The dump is displayed in two portions: a hexadecimal dump (each byte is shown in hexadecimal value) and an ASCII dump (the bytes are shown in ASCII characters). Nonprinting characters are denoted by a period (.) in the ASCII portion of the display. Each display line shows sixteen bytes with a hyphen between the eighth and ninth bytes. At times, displays are split in this manual to fit them on the page. Each displayed line, except possibly the first, begins on a 16-byte boundary.

If you enter the command:

```
dcs:100 110
```

DEBUG displays:

```
04BA:0100 42 45 52 54 41 ... 4E 44 BERTA T. BORLAND
```

If the following command is entered:

```
D
```

the display is formatted as described above. Each line of the display begins with an address; incremented by 16 from the address on the previous line. Each subsequent D (entered without parameters) displays the bytes immediately following those last displayed.

If the user enters the command:

```
DCS:100 L 20
```

the display is formatted as described above, but 20H bytes are displayed.

If the user enters the command:

```
DCS:100 115
```

the display is formatted as described above, but all the bytes in the range of lines from 100H to 115H in the CS segment are displayed.

NAME

Enter

SYNTAX

E<address>[<list>]

FUNCTION

Enter byte values into memory at the specified <address>.

COMMENTS

If the optional <list> of values is entered, the replacement of byte values occurs automatically. (If an error occurs, no byte values are changed.) If the <address> is entered without the optional <list>, DEBUG displays the address and its contents, then repeats the address on the next line and waits for the user's input. At this point, the Enter command waits for you to perform one of the following actions:

1. Replace a byte value with a value the user types in. The user simply types the value after the current value. If the value typed in is not a legal hexadecimal value or if more than two digits are typed, the illegal or extra character is not echoed.

2. Press the space bar to advance to the next byte. To change the value, simply enter the new value as described in (1.) above. If the user spaces beyond an eight-byte boundary, DEBUG starts a new display line with the address displayed at the beginning.

3. Type a hyphen (-) to return to the preceding byte. If the user decides to change a byte behind the current position, typing the hyphen returns the current position to the previous byte. When the hyphen is typed, a new line is started with the address and its byte value displayed.

4. Press the <RETURN> key to terminate the Enter command. The <RETURN> key may be pressed at any byte position.

EXAMPLE

Assume the following command is entered:

```
ECS:100
```

DEBUG displays:

```
04BA:0100  EB._
```

To change this value to 41, enter "41" as shown:

```
04BA:0100  EB.41_
```

To step through the subsequent bytes, press the space bar to see:

```
04BA:0100  EB.41    10.    00.    BC._
```

To change BC to 42:

```
04BA:0100  EB.41    10.    00.    BC.42_
```

Now, realizing that 10 should be 6F; enter the hyphen as many times as needed to return to byte 0101 (value 10), then replace 10 with 6F:

```
04BA:0100  EB.41    10.    00.    BC.42-
04BA:0102  00.-
04BA:0101  10.6F_
```

Pressing the <RETURN> key ends the Enter command and returns to the DEBUG command level.

NAME

Fill

SYNTAX

F<range> <list>

FUNCTION

Fill the addresses in the <range> with the values in the <list>.

COMMENTS

If the <range> contains more bytes than the number of values in the <list>, the <list> will be used repeatedly until all bytes in the <range> are filled. If the <list> contains more values than the number of bytes in the <range>, the extra values in the <list> will be ignored. If any of the memory in the <range> is not valid (bad or nonexistent), the error will be propagated into all succeeding locations.

EXAMPLE

Assume the following command is entered:

```
F04BA:100 L 100 42 45 52 54 41
```

DEBUG fills memory locations 04BA:100 through 04BA:1FF with the bytes specified. The five values are repeated until all 100H bytes are filled.

NAME

Go

SYNTAX

G[=<address>][<address>...]

FUNCTION

Execute the program currently in memory.

COMMENTS

If the Go command is entered alone, the program executes as if the program had run outside DEBUG.

If =<address> is set, execution begins at the address specified. The equal sign (=) is required, so that DEBUG can distinguish the start =<address> from the breakpoint <address>es.

With the other optional addresses set, execution stops at the first <address> encountered, regardless of that address' position in the list of addresses to halt execution, no matter which branch the program takes. When program execution reaches a breakpoint, the registers, flags, and decoded instruction are displayed for the last instruction executed. (The result is the same as if you had entered the Register command for the breakpoint address.)

Up to ten breakpoints may be set. Breakpoints may be set only at addresses containing the first byte of an 8086 opcode. If more than 10 breakpoints are set, DEBUG returns the BP Error message.

The user stack pointer must be valid and have six bytes available for this command. The G command uses an IRET instruction to cause a jump to the program under test. The user stack pointer is set, and the user Flags, Code Segment register, and Instruction Pointer are pushed on the the user stack. (Thus, if the user stack is not valid or is too small, the operating system may crash.) An interrupt code (0CCH) is placed at the specified breakpoint address(es). When an instruction with the breakpoint code is encountered, all breakpoint addresses are restored to their original instructions. If execution is not halted at

one of the breakpoints, the interrupt codes are not replaced with the original instructions.

EXAMPLE

Assume the following command is entered:

```
GCS:7550
```

The program currently in memory executes up to the address 7550 in the CS segment. Then DEBUG displays registers and flags, after which the Go command is terminated.

After a breakpoint has been encounterd, if you enter the Go command again, then the program executes just as if the user had entered the filename at the MS-DOS command level. The only difference is that program execution begins at the instruction after the breakpoint rather than at the usual start address.

NAME

Hex

SYNTAX

H<value> <value>

FUNCTION

Perform hexadecimal arithmetic on the two parameters.

COMMENTS

First, DEBUG adds the two parameters, then subtracts the second parameter from the first. The results of the arithmetic is displayed on one line; first the sum, then the difference.

EXAMPLE

Assume the following command is entered:

```
H19F 10A
```

DEBUG performs the calculations and then returns the results:

```
02A9   0095
```

NAME

Input

SYNTAX

I<value>

FUNCTION

Input and display one byte from the port specified by <value>.

COMMENTS

A 16-bit port address is allowed.

EXAMPLE

Assume the following command is entered:

```
I2F8
```

Assume also that the byte at the port is 42H. DEBUG inputs the byte and displays the value:

```
42
```

NAME

Load

SYNTAX

L[<address> [<drive> <record> <record>]]

FUNCTION

Load a file into memory.

COMMENTS

Set BX:CX to the number of bytes read. The file must have been named either with the DEBUG invocation command or with the N command. Both the invocation and the N commands format a filename properly in the normal format of a file control block at CS:5C.

If the L command is given without any parameters, DEBUG loads the file into memory beginning at address CS:100 and sets BX:CX to the number of bytes loaded. If the L command is given with an address parameter, loading begins at the memory <address> specified. If L is entered with all parameters, absolute disk sectors are loaded, not a file. The <record>s are taken from the <drive> specified (the drive designation is numeric here--0=A:, 1=B:, 2=C:, etc.); DEBUG begins loading with the first <record> specified, and continues until the number of sectors specified in the second <record> have been loaded.

EXAMPLE

Assume the following commands are entered:

```
A:DEBUG
>NFILE.COM
```

Now, to load FILE.COM, enter:

```
L
```

DEBUG loads the file and returns the DEBUG prompt. Assume you want to load only portions of a file or certain records from a disk. To do this, enter:

```
L04ba:100 2 0F 6D
```

DEBUG then loads 109 (6D hex) records beginning with logical record number 15 into memory

beginning at address 04BA:0100. When the records have been loaded, DEBUG simply returns the its prompt.

If the file has a .EXE extension, then it is relocated to the load address specified in the header of the .EXE file: the <address> parameter is always ignored for .EXE files. Note that the header itself is stripped off the .EXE file before it is loaded into memory. Thus the size of a .EXE file on disk will differ from its size in memory.

If the file named by the Name command or specified on invocation is a .HEX file, then entering the L command with no parameters causes loading of the file beginning at the address specified in the .HEX file. If the L command includes the option <address>, DEBUG adds the <address> specified in the L command to the address found in the .HEX file to determine the start address for loading the file.

NAME

Move

SYNTAX

M<range> <address>

FUNCTION

Move the block of memory specified by <range> to the location beginning at the <address> specified.

COMMENTS

Overlapping moves (moves where part of the block overlaps some of the current addresses) are always performed without loss of data. Addresses that could be overwritten are moved first. The sequence for moves from higher addresses to lower addresses is to move the data beginning at the block's lowest address and working towards the highest. The sequence for moves from lower addresses to higher addresses is to move the data beginning at the block's highest address and working towards the lowest.

Note that if the addresses in the block being moved will not have new data written to them, the data there before the move will remain; that is, the M command really copies the data from one area into another, in the sequence described, and writes over the new addresses. This is why the sequence of the move is important.

EXAMPLE

Assume you enter:

```
MCS:100 110 CS:500
```

DEBUG first moves address CS:110 to address CS:510, then CS:10F to CS:50F, and so on until CS:100 is moved to CS:500. You should enter the D command, using the <address> entered for the M command, to review the results of the move.

NAME

Name

SYNTAX

N<filename>[<filename>...]

FUNCTION

Set filenames

COMMENTS

The Name command performs two distinct functions, both having to do with filenames. First, Name is used to assign a filename for a later Load or Write command. Thus, if you invoke DEBUG without naming any file to be debugged, then the N<filename> command must be given before a file can be Loaded. Second, Name is used to assign filename parameters to the file being debugged. In this case, Name accepts a list of parameters that are used by the file being debugged.

These functions overlap. Consider the following set of DEBUG commands:

```
>NFILE1.EXE
>L
>G
```

Because of the two-pronged effect of the Name command, the following happens:

1. (N)ame assigns the filename FILE1.EXE to the filename to be used in any later Load or Write commands.

2. (N)ame also assigns the filename FILE.EXE to the first filename parameter to be used by any program that is later debugged.

3. (L)oad loads FILE.EXE into memory.

4. (G)o causes FILE.EXE to be executed with FILE.EXE as the single filename parameter (that is, FILE.EXE is executed as if FILE FILE.EXE had been typed at the command level.

A more useful chain of commands might go like this:

```
>NFILE1.EXE
>L
>NFILE2.DAT FILE3.DAT
>G
```

Here, Name sets FILE1.EXE as the filename for the subsequent Load command. The Load command loads FILE1.EXE into memory, and then the Name command is used again, this time to specify the parameters to be used by FILE1.EXE. Finally, when the Go command is executed, FILE1.EXE is executed as if FILE1 FILE2.DAT FILE3.DAT had been typed at the MS-DOS command level. Note that if a Write command were executed at this point, then FILE1.EXE--the file being debugged--would be saved with the name FILE2.DAT! To avoid such undesired results, you should always execute a Name command before either a Load or a Write.

There are four distinct regions of memory that can be affected by the Name command:

```
CS:5C  FCB for file 1
CS:6C  FCB for file 2
CS:80  Count of characters
CS:81  All characters entered
```

A File Control Block (FCB) for the first filename parameter given to the Name command is set-up at CS:5C. If a second filename parameter is given, then an FCB is setup for it beginning at CS:6C. The number of characters typed in the Name command (exclusive of the first character, "N") is given at location CS:80. The actual stream of characters given by the Name command (again, exclusive of the letter "N") begins at CS:81. Note that this stream of characters may contain switches and and delimiters that would be legal in any command typed at the MS-DOS command level.

EXAMPLE

A typical use of the Name command would be:

```
DEBUG PROG.COM
-NPARAM1 PARAM2/C
-G
-
```

In this case, the Go command executes the file in memory as if the following command line had been entered:

```
PROG PARAM1 PARAM2/C
```

Testing and debugging therefore reflect a normal runtime environment for PROG.COM.

NAME

Output

SYNTAX

O<value> <byte>

FUNCTION

Send the <byte> specified to the output port specified by <value>.

COMMENTS

A 16-bit port address is allowed.

EXAMPLE

Enter:

```
O2F8 4F
```

DEBUG outputs the byte value 4F to output port 2F8.

NAME

Quit

SYNTAX

Q

FUNCTION

Terminate the debugger.

COMMENTS

The Q command takes no parameters and exits DEBUG without saving the file currently being operated on. You are returned to the MS-DOS commands level.

EXAMPLE

To end the debugging session, enter:

```
Q<RETURN>
```

DEBUG is terminated, and control returns to the MS-DOS command level.

NAME

Register

SYNTAX

R[<register-name>]

FUNCTION

Display the contents of one or more CPU registers.

COMMENTS

If no <register-name> is entered, the R command dumps the register save area and displays the contents of all registers and flags.

If a register name is entered, the 16-bit value of that register is displayed in hexadecimal, and then a colon appears as a prompt. The user then either enters a <value> to change the register, or simply presses the <RETURN> key if no change is wanted.

The only valid <register-name>s are:

AX	BP	SS	
BX	SI	CS	
CX	DI	IP	(IP and PC both refer
DX	DS	PC	to the instruction
SP	ES	F	pointer.)

Any other entry for <register-name> results in a BR Error message.

If F is entered as the <register-name>, DEBUG displays each flag with a two character alphabetic code. To alter any flag, enter the opposite two letter code. The flags are either set or clear.

The flags with their codes for set and clear are listed below:

FLAG NAME	SET	CLEAR
Overflow	OV	NV
Direction	DN Decrement	UP Increment
Interrupt	EI Enabled	DI Disabled
Sign	NG Negative	PL Plus
Zero	ZR	NZ
Auxiliary Carry	AC	NA
Parity	PE Even	PO Odd
Carry	CY	NC

Whenever the user enters the command RF, the flags are displayed in the order shown above in a row at the beginning of a line. At the end of the list of flags, DEBUG displays a hyphen (-). You may enter new flag values as alphabetic pairs. The new flag values can be entered in any order. You are not required to leave spaces between the flag entries. To exit the R command, press the <RETURN> key. Flags for which new values were not entered remain unchanged.

If more than one value is entered for a flag, DEBUG returns a DF Error message. If the you enter a flag code other than those shown above, DEBUG returns a BF Error message. In both cases, the flags up to the error in the list are changed; flags at and after the error are not.

At start up, the segment registers are set to the bottom of free memory, the Instruction Pointer is set to 0100H, all flags are cleared, and the remaining registers are set to zero.

EXAMPLE

Enter:

```
R
```

DEBUG displays all registers, flags, and the decoded instruction for the current location. If the location is CS:11A, then DEBUG might display:

```
AX=0E00 BX=00FF CX=0007 DX=01FF SP=039D BP=0000
SI=005C DI=0000 DS=04BA ES=04BA SS=04BA CS=04BA
IP=011A  NV UP DI NG NZ AC PE NC
04BA:011A  CD21          INT     21
```

If you enter:

```
RF
```

DEBUG displays the flags:

```
NV UP DI NG NZ AC PE NC - _
```

Now enter any valid flag designation, in any order, with or without spaces.

For example:

```
NV UP DI NG NZ AC PE NC - PLEICY<RETURN>
```

DEBUG responds only with the DEBUG prompt. To see the changes, enter either the R or RF command:

```
RF
NV UP EI PL NZ AC PE CY - _
```

Press <RETURN> to leave the flags this way, or to enter different flag values.

NAME

Search

SYNTAX

S<range> <list>

FUNCTION

Search the range specified for the <list> of bytes specified.

COMMENTS

The <list> may contain one or more bytes, each separated by a space or comma. If the <list> contains more than one byte, only the first address of the byte string is returned. If the <list> contains only one byte, all addresses of the byte in the <range> are displayed.

EXAMPLE

If you enter:

```
SCS:100 110 41
```

DEBUG might return the response:

```
04BA:0104
04BA:010D
>_
```

NAME

Trace

SYNTAX

T[=<address>][<value>]

FUNCTION

Execute one instruction and display the contents of all registers, flags, and the decoded instruction.

COMMENTS

If the optional =<address> is entered, tracing occurs at the =<address> specified. The optional <value> causes DEBUG to execute and trace the number of steps specified by <value>.

The T command uses the hardware trace mode of the 8086 or 8088 microprocessor. Consequently, the user may also trace instructions stored in ROM.

EXAMPLE

Enter:

```
T
```

DEBUG returns a display of the registers, flags, and decoded instruction for that one instruction. Assume that the current position is 04BA:011A; then DEBUG might return the display:

```
AX=0E00 BX=00FF CX=0007 DX=01FF SP=039D BP=0000
SI=005C DI=0000 DS=04BA ES=04BA SS=04BA CS=04BA
IP=011A  NV UP DI NG NZ AC PE NC
04BA:011A  CD21           INT     21
```

Now enter:

```
T=011A 10
```

DEBUG executes sixteen (10 hex) instructions beginning at 011A in the current segment and then displays all registers and flags for each instruction as it is executed. The display scrolls away until the last instruction is executed. Then the display stops, and you can see the register and flag values for the last few instructions performed. Remember that

<CONTROL-S> suspends the display at any point, so that you can study the registers and flags for any instruction.

NAME

Unassemble

SYNTAX

U[<range>]

FUNCTION

Disassemble bytes and display the source statements that correspond to them, along with addresses and byte values.

COMMENTS

The display of disassembled code looks like a listing for an assembled file. If you enter the U command without parameters, 20 hexadecimal bytes are disassembled at the first address after that displayed by the previous Unassemble command. If you enter the U command with the <range> parameter, then DEBUG disassembles all bytes in the range. If the <range> is given as an <address only, then 20H bytes are disassembled, not 80H.

EXAMPLE

Enter:

```
U04BA:100 L10
```

DEBUG disassembles 16 bytes beginning at address 04BA:0100:

```
04BA:0100  206472     AND   [SI+72],AH
04BA:0103  69         DB    69
04BA:0104  7665       JBE   016B
04BA:0106  207370     AND   [BP+DI+70],DH
04BA:0109  65         DB    65
04BA:010A  63         DB    63
04BA:010B  69         DB    69
04BA:010C  66         DB    66
04BA:010D  69         DB    69
04BA:010E  63         DB    63
04BA:010F  61         DB    61
```

If you enter:

```
u04ba:0100 0108
```

The display shows:

```
04BA:0100  206472      AND   [SI+72],AH
04BA:0103  69          DB    69
04BA:0104  7665        JBE   016B
04BA:0106  207370      AND   [BP+DI+70],DH
```

If the bytes in some addresses are altered, the disassembler alters the instruction statements. The U command can be entered for the changed locations, the new instructions viewed, and the disassembled code used to edit the source file.

NAME

Write

SYNTAX

W[<address>[<drive> <record> <record>]]

FUNCTION

Write the file being debugged to a disk file.

COMMENTS

If only the W appears, BX:CX must already be set to the number of bytes to be written; the file is written beginning from CS:100. If the W command is given with just an address, then the file is written beginning at that address. If a G or T command was used, BX:CX must be reset before using the Write command without parameters. (Note that if a file is loaded and modified, the name, length, and starting address are all set correctly to save the modified file as long as the length has not changed.)

The file must have been named either with the DEBUG invocation command or with the N command (see Name above). Both the invocation and the N commands format a file name properly in the normal format of a file control block at CS:5C.

If the W command is given with parameters, the write begins from the memory address specified; the file is written to the <drive> specified (the drive designation is numeric here--0=A:, 1=B:, 2=C:, etc.); DEBUG writes the file beginning at the logical record number specified by the first <record>; and continues until the number of sectors specified in the second <record> have been written.

WARNING

Writing to absolute sectors is EXTREMELY dangerous because the process bypasses the file handler.

EXAMPLE

Enter:

```
W
```

DEBUG writes out the file to disk then displays the DEBUG prompt:

```
W
>_
```

Another example:

```
WCS:100 1 37 2B
```

DEBUG writes out the contents of memory, beginning with the address CS:100 to the disk in drive B:. The data written out starts in disk logical record number 37H and consists of 2BH records. When the write is complete, DEBUG displays the prompt:

```
WCS:100 1 37 2B
>_
```

5.4 ERROR MESSAGES

During the DEBUG session, you may receive any of the following error messages. Each error terminates the DEBUG command with which it is associated, but does not terminate DEBUG itself.

ERROR CODE	DEFINITION
BF	Bad Flag The user attempted to alter a flag, but the characters entered were not one of the acceptable pairs of flag values. See the Register command for the list of acceptable flag entries.
BP	Too many Breakpoints The user specified more than ten breakpoints as parameters to the G command. Reenter the Go command with ten or fewer breakpoints.
BR	Bad Register The user entered the R command with an invalid register name. See the Register command for the list of valid register names.
DF	Double Flag The user entered a two values for one flag. The user may specify a flag value only once per RF command.

CHAPTER 6

FILCOM

The FILCOM File Comparison Utility compares the contents of two files. The differences between the two files are output to a third file. The files being compared may be either source files (files containing source statements of a programming language) or binary files (files output by the MACRO-86 assembler, the MS-LINK Linker Utility, or by one of the Microsoft high-level language compilers).

Limitations on Source Comparisons

FILCOM uses all available memory as buffer space to hold the source files. If the source files are larger than available memory, FILCOM comparisons what it is able to load into the buffer space. If no matches are found within the portions of the files in the buffer space, FILCOM outputs only the message:

FILES ARE DIFFERENT

For binary files larger than available memory, FILCOM compares both files completely, overlaying the portion in memory with the next portion from disk. All differences are output the same as for binary comparisons of files that fit completely in memory.

6.1 INVOCATION

FILCOM can be invoked in one of two ways.

Method 1:

Enter:

```
FILCOM
```

FILCOM responds with the first prompt and then waits:

```
Source 1 Filename [.ASM]:
```

Method 2:

Enter:

```
FILCOM <sourcel>,[<source2>],[<list>][/<switch>...]
```

FILCOM and the filenames must be separated by commas. The slash mark is the only delimiter allowed between a filename and a switch letter. Switches may be placed after any of the entries in the invocation command line, but before the comma.

If you want to select the default for Source 2 but not for List, enter two consecutive commas between Source 1 and List entries.

For example:

```
FILCOM ALPHA,,GAMMA
```

When method 2 is used, FILCOM responds with a banner but no prompts, and performs the comparison. Method 2 permits FILCOM commands to be used in a batch file under MS-DOS, as well as permitting you to enter all commands on one line at one time. When FILCOM is finished, the operating system prompt reappears.

6.2 COMMANDS

Commands to FILCOM consist of responses to three prompts for file specifications, plus optional switches. The file specifications may be entered one at a time as the prompts appear, or all at once as part of the FILCOM invocation command (method 2).

6.2.1 File Specifications

All file specifications take the form:

```
d:filename.ext
```

where: d: is the letter of a disk drive. If the drive designation is omitted, FILCOM defaults to the operating system's (current) default drive.

filename is a 1-8 character name of the file.

.ext is a 1-3 character extension to the filename. See Section 5.2.3, Defaults and Shortcuts, for a list of the default filename extensions used under FILCOM and their effects.

6.2.2 Prompts

If invocation method 2 is used, FILCOM displays no prompts; it simply performs the comparison and exits to the operating system.

If invocation method 1 is used (or else method 2 with an illegal filename or the name of a nonexistent file for the first source file), FILCOM displays the first prompt:

```
Source 1 Filename [.ASM]:_
```

Enter the name of one of the files you want compared. If the filename extension for this file is .ASM, the extension may be omitted from the entry. Otherwise, the extension must be included.

When a legal response has been entered, FILCOM responds with the second prompt:

```
Source 2 Filename [source1.BAK]:_
```

Enter the name of the file you want compared to Source 1. FILCOM defaults to the backup file for the file named for the first prompt. If the response to prompt 1 is TEST (meaning TEST.ASM), FILCOM will display as default for Source 2 the filename TEST.BAK. If you want to compare the Source 1 file with its backup file, simply press <RETURN>. Otherwise, enter a filename. Likewise, if the Source 2 file has a filename extension of .BAK, the extension may be omitted. Otherwise, the extension must be entered, too.

When a legal response to the second prompt has been entered, FILCOM responds with the third prompt:

```
List Filename [source1.DIF]:_
```

Enter the name of the file to receive the list of differences. FILCOM defaults to the name given for Source 1 with a default filename extension of .DIF. Again, if the response to prompt 1 was TEST (meaning TEST.ASM), FILCOM displays TEST.DIF as the default List filename. If this default filename is acceptable to you for the List file, simply press the carriage return key. Otherwise, enter the filename. Likewise, if the filename extension .DIF is acceptable to you, the extension may be omitted, even if you do specify a filename. If .DIF is an unacceptable filename extension, enter an extension along with the filename.

When FILCOM is finished comparing the two source files and has output the differences to the List file, the operating system prompt reappears.

6.2.3 Defaults And Shortcuts

FILCOM recognizes the following default extensions:

Prompt	Extension	Effect
Source 1	.ASM	Default Source 1 filename extension. May be overridden.
	.OBJ .EXE .COM	Causes default to binary comparison
Source 2	.BAK	Default Source 2 Filename extension. May be overridden.
List	.DIF	Default List Filename extension. May be overridden.

Shortcuts

Two shortcuts for entering commands are supported. Both shortcuts use default responses for any prompts to which a response is not entered.

Carriage return key

The Source 1 Filename [.ASM] prompt requires at least a filename response.

The Source 2 Filename [source1.BAK] and List Filename [source1.DIF] prompts show a default entry; the filename entered for Source 1 and a default filename extension. To select the default entry, simply press the <RETURN> key.

Example:

```
Source 1 Filename [.ASM]: TEST
Source 2 Filename [TEST.BAK]: <RETURN>
List Filename [TEST.DIF]: <RETURN>
```

These responses cause FILCOM to compare TEST.ASM with TEST.BAK and to output any differences in the file TEST.DIF.

<RETURN> only may be entered for either of the prompts, regardless of what you plan to enter for the other. For example, the <RETURN> may be used to select the default for Source 2, yet allows a nondefault entry for List.

Example:

```
Source 1 Filename [.ASM]: TEST
Source 2 Filename [TEST.BAK]: <RETURN>
List Filename [TEST.DIF]: PAST.PRN
```

These responses cause FILCOM to compare TEST.ASM with TEST.BAK and to output any differences in the file PAST.PRN (default for Source 2 was selected, but not for List.)

Semicolon (;)

The semicolon character (;) also selects the default responses to the Source 2 and List prompts. If the semicolon is entered following the Source 2 Filename prompt, the List Filename prompt will not appear. That is, the semicolon selects the default response for all remaining prompts. Unlike the carriage return key, once you enter semicolon, comparing begins, and you have no chance to enter another response for that comparison. Indeed, you may think of the semicolon as a message to FILCOM that you want to use all default responses only.

This is especially useful when using invocation method 2:

Example:

```
FILCOM TEST;
```

This entry causes FILCOM to display its banner and then perform the desired comparison. FILCOM compares TEST.ASM with TEST.BAK and outputs the differences in the file TEST.DIF (the same as the example under "Carriage Return Key" above).

Now, consider this sample invocation:

```
FILCOM
Source 1 Filename [.ASM]: TEST;
```

This set of commands and responses produces the same result as the two previous examples. Note that you have no chance to enter alternatives to the defaults for Source 2 or List when you use the semicolon.

6.2.4 Switches

FILCOM supports five switches. Switches are single letters appended to the (method 2) invocation command line or to any of the prompt responses to control the file comparison. A switch must always be preceded by a slash.

FILCOM switches are one of two types: source comparison or binary comparison.

Source Comparison Switches

/A Force a source comparison of files with filename extensions .OBJ, .EXE, and .COM. FILCOM defaults to binary comparison on files with these filename extensions. Files with any other filename extensions default to source comparison. Therefore, /A is not required for files that do not have one of these three filename extensions.

/C Include comments in comparison. A comment starts with a semicolon (;), and ends with an end-of-line character. By default, comments are not included in comparisons. Thus, only functional changes in source files are detected by FILCOM. This means that comments in two files, even if different, are ignored when searching for consecutive lines that match (see the /<n> switch below for an explanation of "match").

/<n> <n> is a number from 1 through 9. Default is 3. <n> specifies how many consecutive lines in the two files must be the same before FILCOM considers that the two files match at that point. (Refer to examples 1 and 2 below for a demonstration of the effects of the /<n> switch).

When FILCOM finds <n> lines that match, it outputs the lines that are different since the last <n> lines that matched, plus the first line of the current <n> lines that match. The first match line should help locate where differences occurred.

/S Include spaces and tabs in comparisons. By default, spaces and tabs are <u>not</u> included in comparisons. Thus, only functional changes in source files are detected by FILCOM. This means that spaces and tabs in the two files, even if the same, are ignored and are not used to find matches.

Binary Comparison Switch

/B Force binary comparison of files that default to source comparison (files without the filename extensions .OBJ, .EXE, or .COM). Instead of a source comparison of lines, FILCOM compares the two files byte-by-byte. For differences, FILCOM outputs the offset location into the files and the differing bytes in hexadecimal. (Refer to example 3 for a demonstration.)

6.3 EXAMPLES

Example 1

Assume these two ASCII files are on disk:

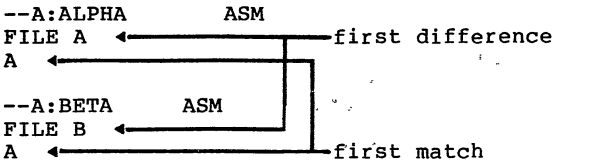
ALPHA.ASM	BETA.ASM
FILE A	FILE B
A	A
B	B
C	C
D	G
E	H
F	I
G	J
H	1
I	2
M	P
N	Q
O	R
P	S
Q	T
R	U
S	V
T	4
U	5
V	W
W	X
X	Y
Y	Z
Z	

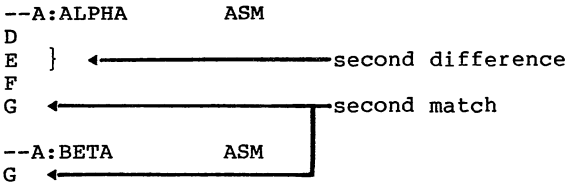
To compare the two files and output the differences on the terminal screen, enter the following (method 2) command line:

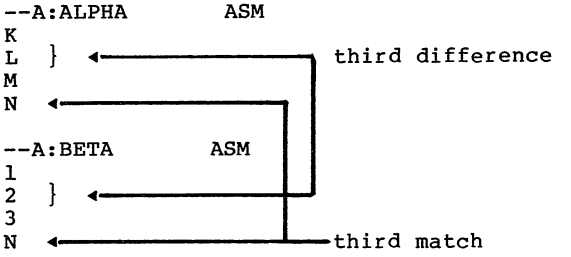
```
FILCOM ALPHA,BETA.ASM,CON
```

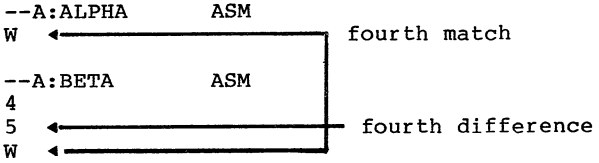
FILCOM is directed to compare ALPHA.ASM with BETA.ASM and output the differences on the terminal screen. All other defaults remain intact (do not use tabs, spaces, or comments for matches, and conduct a source comparison).

The output appears as follows on the terminal screen:


--A:ALPHA ASM
FILE A
first difference
A
--A:BETA ASM
FILE B
A
first match


--A:ALPHA ASM
D
E
second difference
F
G
second match
--A:BETA ASM
G


--A:ALPHA ASM
K
L
third difference
M
N
--A:BETA ASM
1
2
3
N
third match


--A:ALPHA ASM
W
fourth match
--A:BETA ASM
4
5
fourth difference
W

Example 2

Using the same two source files, output the differences on the line printer. Also, require that four successive lines must be the same to constitute a match.

Using invoke method 2 again, enter:

```
FILCOM ALPHA,BETA.ASM,PRN/4
```

The following output should appear on the line printer:

```
--A:ALPHA        ASM
FILE A
A
B   }
C
D
E
F
G

--A:BETA         ASM
FILE B
A
B   }
C
G
```

These lines are listed as different because the /4 switch specifies that 4 consecutive lines must be found identical in the two files before they are considered a match.

```
------------------------

--A:ALPHA        ASM
K
L
M
N

--A:BETA         ASM
1
2
3
N

------------------------

--A:ALPHA        ASM
W

--A:BETA         ASM
4
5
W
```

Example 3

Using the same two source files again, force a binary comparison, then output the differences on the terminal screen.

Using invoke method 1,enter:

```
FILCOM
```

FILCOM responds:

```
Source 1 Filename [.ASM]:
```

Entries and responses should appear as follows:

```
Source 1 Filename [.ASM]:  ALPHA/B
Source 2 Filename [ALPHA.BAK]:  BETA.ASM
List Filename [ALPHA.DIF]:  CON
```

The /B switch at the end of the Source 1 line forces binary comparison. This switch, and the others, may be entered at the end of any of the response entries. The switches may be, but need not be, entered on the same response line.

The screen display should appear as follows:

```
ADDRESS ALPHA ASM       BETA ASM
 00006   41      42
 00012   44      47
 00015   45      48
 00018   46      49
 0001B   47      4A
 0001E   48      31
 00021   49      32
 00024   4A      33
 00027   4B      4E
 0002A   4C      4F
 0002D   4D      50
 00030   4E      51
 00033   4F      52
 00036   50      53
 00039   51      54
 0003C   52      55
 0003F   53      56
 00042   54      34
 00045   55      35
 00048   56      57
 0004B   57      58
 0004E   58      59
 00051   59      5A
```

APPENDIX A

Instructions for Single Disk Drive Users

For single disk drive users the commands are exactly the same syntax as for two drive users. The difference lies in your perception of the "arrangement" of the drives.

You must think of his system as having two disk drives: drive A: and drive B:. However, instead of A: and B: designating physical disk drive mechanisms, the A: and B: designate diskettes. Therefore, when the user specifies drive B: while operating on drive A: (the prompt is A:), MS-DOS prompts the user to "switch drives" by swapping diskettes.

The prompts are:

> Insert diskette for drive A: and strike any key when ready
>
> Insert diskette for drive B: and strike any key when ready

These procedures apply to any MS-DOS COMMAND commands (both internal and external) that can request or direct a different drive as a part of its syntax. These commands include:

```
CHKDSK [d:]
COPY <filespec>[filespec]
DEL <filespec> [filespec...]
DIR [d:][filename]
FORMAT d:[/S]
RENAME <filespec> <filename>
TYPE <filespec>
```

Also, if any of these commands are used in a BATCH file and call for a different drive, the single disk drive procedures apply. Execution is halted and the appropriate prompt is displayed.

EXAMPLE

The following example may serve as an illustration for all of the commands listed above:

```
A:COPY COMMAND.COM B:  Insert diskette for drive
B: and depress space bar when ready
    1 File(s) copied

Insert diskette for drive A: and depress space bar
when ready
A:_
```

APPENDIX B

MS-DOS File Control Block Definition

The MS-DOS File Control Block (FCB) is defined as follows:

byte 0	Drive Code. Zero specifies the default drive, 1=drive A, 2=drive B, etc.
bytes 1-8	File name. If the file is less than 8 characters, the name must be left justified with trailing blanks.
bytes 9-11	Extension to file name. If less than 3 characters, must be left justified with trailing blanks. May also be all blanks.
bytes 12-13	Current block (extent). This word (low byte first) specifies the current block of 128 records, relative to the start of the file, in which sequential disk reads and writes occur. If zero, then the first block of the file is being accessed; if one, then the second etc. Combined with the current record field(byte 32) a particular logical record is identified.
bytes 14-15	Size of the record the user wishes to work with. This word may be filled immediately after an OPEN of the file if the default logical record size(128 bytes) is not desired. The Open and Create functions set this field to 128; it is also changed to 128 if a read or write is attempted with the field set to zero.
bytes 16-19	File size. This is the current size, in bytes, of the file. It may read by user programs but must not be written by them.
bytes 20-21	Date. This is normally the date of the last write to the file. It is set by all disk write operations and Create to today's date. It is set by Open to the date recorded in the

disk directory for the file. User programs may modify this field after writing to a file but before closing it to change the date recorded in the disk directory.

The format of this 16-bit field is as follows: bits 0-4, day of month; bits 5-8, month of year; bits 9-15, year minus 1980. All zero means no date.

bytes 22-23 Time. Similar to Date, above. The format is bits 0-4, seconds/2; bits 5-10, minutes; bits 11-15, hours.

bytes 24-31 Reserved for MS-DOS.

byte 32 Current record. Identifies the record within the current block of 128 records that will be accessed with a sequential read or write function. See Current Block,bytes 12-13.

bytes 33-36 Random Record. This field must only be set if the file is to be accessed with a random read or write function. If the record size is greater than or equal to 64 bytes, only the first 3 bytes are used, as a 24-bit number representing the position in the file of a record. If the record size if less than 64 bytes, all 4 bytes are used as 32-bit number of the same purpose. This field is thus large enough to address any byte in a file of the maximum size, 2^30 bytes.

THE EXTENDED FCB

The extended FCB is a special format used to search for files in the disk directory with special attributes. It consists of 7 bytes in front of a normal FCB, formatted as follows:

FCB-7 Flag. FF hex is placed here to signal an extended FCB.

FCB-6 - FCB-2 Zero field

FCB-1 Attribute byte. If bit 1 = 1, hidden files will be included in directory searches. If bit 2 = 1, system files will be included in directory searches.

Any reference in the description of MS-DOS function calls to an FCB, whether opened or unopened, may use either a normal FCB or an extended FCB. A normal FCB has the same effect as an extended FCB with the attribute byte set to zero.

APPENDIX C

MS-DOS INTERRUPTS AND FUNCTION CALLS

C.1 INTERRUPTS

MS-DOS Reserves interrupt types 20 to 3F hex for its use. This means absolute locations 80 to FF hex are the transfer address storage locations reserved by the DOS. The defined interrupts are as follows with all values in hex:

20 Program terminate. This is the normal way to exit a program. This vector transfers to the logic in the DOS for restoration of <CONTROL-C> exit addresses to the values they had on entry to the program. All file buffers are flushed to disk. All files that have changed in length should have been closed (see function call 10 hex) prior to issuing this interrupt. If the changed file was not closed its length will not be recorded correctly in the directory. When this interrupt is executed, CS MUST point to the 100H parameter area.

21 Function request. See section II FUNCTION REQUESTS.

22 Terminate address. The address represented by this interrupt(88-8B hex) is the the address to which control will transfer when the program terminates. This address is copied into low memory of the segment the program is loaded into at the time this segment is created. If a program wishes to execute a second program it must set the terminate address prior to creation of the segment the program will be loaded into. Otherwise once the second program executes its termination would cause transfer to its host's termination address.

23 <CONTROL-C> exit address. If the user types <CONTROL-C> during keyboard input or video output, "^C" will be printed on the console and an interrupt type 23 hex will be executed. If the <CONTROL-C>

routine preserves all registers, it may end with a a return-from-interrupt instruction (IRET) to continue program execution. If functions 9 or 10 (buffered output and input), were being executed, then I/O will continue from the start of the line. When the interrupt occurs, all registers are set to the value they had when the original call to MS-DOS was made. There are no restrictions on what the <CONTROL-C> handler is allowed to do, including MS-DOS function calls, so long as the registers are unchanged if IRET is used.

If the program creates a new segment, loads in a second program which itself changes the <CONTROL-C> address, the termination of the second program and return to the first will cause the <CONTROL-C> address to be restored to the value it had before execution of the second program.

24 Fatal error abort vector. When a fatal error occurs within MS-DOS, control will be transferred with an INT 24H. On entry to the error handler, AH will have its bit 7=0 if the error was a hard disk error (probably the most common occurrence), bit 7=1 if not. If it is a hard disk error, bits 0-2 include the following:

bit 0			0 if read, 1 if write
bit 2	1		AFFECTED DISK AREA
0	0		Reserved area
0	1		File allocation table
1	0		Directory
1	1		Data area

AL, CX, DX, and DS:BX will be setup to perform a retry of the transfer with INT 25H or INT 26H (below). DI will have a 16-bit error code returned by the hardware.

The values returned are:

0	write protect
2	disk not ready
4	data error
6	Seek error
8	Sector not found
A	Write fault
C	General disk failure

The registers will be set up for a BIOS disk call and the returned code will be in the lower half of the DI register with the upper half undefined. The user stack will look as follows from top to bottom:

```
IP      Registers such that if an IRET is executed
CS      the DOS will respond according to (AL)
FLAGS   as follows:

        (AL)=0      ignore the error
            =1      retry the operation
                    (IF THIS OPTION USED STACK DS,
                     BX,CX AND DX MUST NOT BE MODIFIED!)
            =2 abort the program


AX      USER REGISTERS AT TIME OF REQUEST
BX
CX
DX
SI
DI
BP
DS
ES
IP      The interrupt from the user to the DOS
CS
FLAGS
```

Currently, the only error possible when AH bit 7=1 is a bad memory image of the file allocation table.

Absolute disk read. This transfers control directly to the DOS BIOS. Upon return, the original flags are still on the stack (put there by the INT instruction). This is necessary because return information is passed back in the flags. Be sure to to pop the stack to prevent uncontrolled growth. For this entry point "records" and "sectors" are the same size. The request is as follows:

```
(AL)     Drive number (0=A, 1=B, etc.)
(CX)     Number of sectors to read
(DX)     Beginning logical record number
(DS:BX)  Transfer address
```

The number of records specified are transferred between the given drive and the transfer address. "Logical record numbers" are obtained by numbering each sector sequentially starting from zero and continuing across track boundarys. For example logical record number 0 is track 0 sector 1, whereas logical record number 12 hex is track 2 sector 3.

All registers but the segment registers are destroyed by this call. If the transfer was successful the carry flag (CF) will be zero. If the transfer was not successful CF=1 and (AL) will indicate the error as follows:

Return	Description
0	Attempt to write on write protected disk
2	Disk not ready
4	Data error
6	Seek error
8	Sector not found
C	General disk failure
A	Write fault

26 Absolute disk write. This vector is the counterpart to interrupt 25 above. Except for the fact that this is a write the description above applies.

27 Terminate but stay resident. This vector is used by programs which are to remain resident when COMMAND regains control. Such a program is loaded as an executing COM file by COMMAND. After it has initialized itself, it must set DX to its last address plus one in the segment it is executing in, then execute an interrupt 27H. COMMAND will then treat the program as an extension of MS-DOS, and the program will not be overlaid when other programs are executed.

C.2 FUNCTION REQUESTS

The user requests a function by placing a function number in the AH register, supplying additional information in other registers as necessary for the specific function then executing an interrupt type 21 hex. When MS-DOS takes control it switches to an internal stack. User registers except AX are preserved unless information is passed back to the requester as indicated in the specific requests. The user stack needs to be sufficient to accommodate the interrupt system. It is recommended that it be 80 hex in addition to the user needs. There is an additional mechanism provided for programs that conforms to CP/M calling conventions. The function number is placed in the CL register, other registers are set as normal according to the function specification, and an intrasegment call is made to location 5 in the current code segment. This method is only available to functions which do not pass a parameter in AL and whose numbers are equal to or less than 36. Register AX is always destroyed if this mechanism is used, otherwise it is the same as normal function requests. The functions are as follows with all values in hex:

0 Program terminate. The terminate and <CONTROL-C> exit addresses are restored to the values they had on entry to the terminating program. All file

buffers are flushed, but files which have been changed in length but not closed will not be recorded properly in the disk directory. Control transfers to the terminate address.

1 Keyboard input. Waits for a character to be typed at the keyboard, then echos the character to the video device and returns it in AL. The character is checked for a <CONTROL-C>. If this key is detected an interrupt 23 hex will be executed.

2 Video output. The character in DL is output to the video device. If a <CONTROL-C> is detected after the output an interrupt 23 hex will be executed.

3 Auxiliary input. Waits for a character from the auxiliary input device, then returns that character in AL.

4 Auxiliary output. The character in DL is output to the auxiliary device.

5 Printer output. The character in DL is output to the printer.

6 Direct console I/O. If DL is FF hex, the AL returns With keyboard input character if one is ready, otherwise 00. If DL is not FF hex, then DL is assumed to have a valid character which is output to the video device.

7 Direct console input. Waits for a character to be typed at the keyboard, then returns the character in AL. As with function 6, no checks are made on the character.

8 Console input without echo. This function is identical to function 1, except the key is not echoed.

9 Print string. On entry, DS:DX must point to a character string in memory terminated by a "$" (24 hex). Each character in the string will be output to the video device in the same form as function 2.

A Buffered keyboard input. On entry, DS:DX point to an input buffer. The first byte must not be zero and specifies the number of characters the buffer can hold. Characters are read from the keyboard and placed in the buffer beginning at the third byte. Reading the keyboard and filling the buffer continues until <RETURN> is typed. If the buffer fills to one less than the maximum, then additional keyboard input is ignored until a <RETURN> is typed. The second byte of the buffer is set to the number

of characters received excluding the carriage return (0D hex), which is always the last character. Editing of this buffer is described in the main MS-DOS document under "template editing".

B Check keyboard status. If a character is available from the keyboard, AL will be FF hex, Otherwise AL will be 00.

C Character input with buffer flush. First the keyboard type-ahead buffer is emptied. Then if AL is 1, 6, 7, 8, or 0A hex, the corresponding MS-DOS input function is executed. If AL is not one of these values, no further operation is done and AL returns 00.

D Disk reset. Flushes all file buffers. Unclosed files that have been changed in size will not be properly recorded in the disk directory until they are closed. This function need not be called before a disk change if all files which have been written have been closed.

E Select disk. The drive specified in DL (0=A, 1=B, etc) is selected as the default disk. The number of drives is returned in AL.

F Open file. On entry, DS:DX point to an unopened file control block (FCB). The disk directory is searched for the named file and AL returns FF hex if it is not found. If it is found, AL will return a 00 and the FCB is filled as follows:

If the drive code was 0 (default disk), it is changed to actual disk used (A=1, B=2, etc.) This allows changing the default disk without interfering with subsequent operations on this file. The high byte of the current block field is set to zero. The size of the record to be worked with(FCB bytes E-F hex) is set to the system default of 80 hex. The size of the file, and the time and date are set in the FCB from information obtained from the directory.

It is the user's responsibility to set the record size (FCB bytes E-F) to the size he wishes to think of the file in terms of, if the default 80 hex is not appropriate. It is also the user's responsibility to set the random record field and/or current block and record fields.

10 Close file. This function must be called after file writes to insure all directory information is updated. On entry, DS:DX point to an opened FCB. The disk directory is searched and if the file is

found, its position is compared with that kept in the FCB. If the file is not found in the directory, it is assumed the disk has been changed and AL returns FF hex. Otherwise, the directory is updated to reflect the status in the FCB and AL returns 00.

11 Search for the first entry. On entry, DS:DX point to an unopened FCB. The disk directory is searched for the first matching name (name could have "?"'s indicating any letter matched) and if none are found AL returns FF hex. Otherwise, locations at the disk transfer address are set as follows:

1. If the FCB provided for searching was an extended FCB, then the first byte is set to FF hex, then 5 bytes of zeros, then the attribute byte from the search FCB, then the drive number used (A=1, B=2, etc.), then the 32 bytes of the directory entry. Thus the disk transfer address contains a valid unopened extended FCB with the same search attributes as the search FCB.

2. If the FCB provided for searching was a normal FCB, then the first byte is set to the drive number used (A=1, B=2, etc.) and the next 32 bytes contain the matching directory entry. Thus the disk transfer address contains a valid unopened normal FCB.

Directory entries are formatted as follows:

Location	Bytes	Description
0	11	File name and extension
11	1	Attributes. Bits 1 or 2 make file hidden
10	10	Zero field (for expansion)
22	2	Time. Bits 0-4 = secs/2 5-10 = min 11-15 = hrs
24	2	Date. Bits 0-4 = day 5-8 = month 9-15 = year
26	2	First allocation unit
28	4	File size, in bytes. (30 bits max.)

12 Search for the next entry. After function 11 has been called and found a match, function 12 may be called to find the next match to an ambiguous request("?"'s in the search filename). Both inputs and outputs are are the same as function 11. The reserved area of the FCB keeps information necessary for continuing the search, so it must not be modified.

13 Delete file. On entry, DS:DX point to an unopened FCB. All matching directory entries are deleted. If no directory entries match, AL returns FF, otherwise AL returns 00.

14 Sequential read. On entry, DS:DX point to an opened FCB. The record addressed by the current block (FCB bytes C-D) and the current record(FCB byte 1F) is loaded at the disk transfer address, then the record address is incremented. If end-of-file is encountered AL returns either 01 or 03. A return of 01 indicates no data in the record, 03 indicates a partial record is read and filled out with zeros. A return of 02 means there was not enough room in the disk transfer segment to read one record, so the transfer was aborted. AL returns 00 if the transfer was completed successfully.

15 Sequential write. On entry, DS:DX point to an opened FCB. The record addressed by the current block and current record fields is written from the disk transfer address(or in the case of records less than sector sizes is buffered up for an eventual write when a sector's worth of data is accumulated). The record address is then incremented. If the disk is full AL returns with a 01. A return of 02 means there was not enough room in the disk transfer segment to write one record, so the transfer was aborted. AL returns 00 if the transfer was completed successfully.

16 Create file. On entry DS:DX point to an unopened FCB. The disk directory is searched for an empty entry, and AL returns FF if none is found. Otherwise, the entry is initialized to a zero-length file, the file is opened(see function F), and AL returns 00.

17 Rename file. On entry, DS:DX point to a modified FCB which has a drive code and file name in the usual position, and a second file name starting 6 bytes after the first(DS:DX+11 hex) in what is normally a reserved area. Every matching occurrence of the first is changed to the second(with the restriction that two files cannot have the exact same name and extension). If "?"'s appear in the second name, then the corresponding positions in the original name will be unchanged. AL returns FF hex if no match was found, otherwise 00.

19 Current disk. AL returns with the code of the current default drive (0=A, 1=B, etc.)

1A Set disk transfer address. The disk transfer address is set to DS:DX. MS-DOS will not allow disk transfers to wrap around within the segment, nor to overflow into the next segment.

1B Allocation table address. On return, DS:BX point to the allocation table for the current drive, DX has the number of allocation units, and AL has the number of records per allocation unit, and CX has the size of the physical sector. At DS:[BX-1], the byte before the allocation table, is the dirty byte for the table. If set to 01, it means the table has been modified and must be written back to disk. If 00, the table is not modified. Any programs which get the address and directly modify the table must be sure to set this byte to 01 for the changes to be recorded. This byte should NEVER be set to 00 - instead, a DISK RESET function (#0D hex) should be performed to write the table and reset the bit.

21 Random read. On entry, DS:DX point to an opened FCB The current block and current record are set to agree with the random record field, then the record addressed by these fields is loaded at the current disk transfer address. If end-of-file is encountered, AL returns either 01 or 03. If 01 is returned no more data is available. If 03 is returned, a partial record is available, filled out with zeros. A return of 02 means there was not enough room in the disk transfer segment to read one record, so the transfer was aborted. AL returns 00 if the transfer was completed successfully.

22 Random write. On entry, DS:DX point to an opened FCB. The current block and current record are set to agree with the random record field, then the record addressed by these fields is written (or in the case of records not the same as sector sizes -buffered) from the disk transfer address. If the disk is full AL returns 01. A return of 02 means there was not enough room in the disk transfer segment to write one record, so the transfer was aborted. AL returns 00 if the transfer was completed successfully.

23 File size. On entry, DS:DX point to an unopened FCB. The disk directory is searched for the first matching entry and if none is found, AL returns FF. Otherwise the random record field is set with the size of the file(in terms of the record size field rounded up) and AL returns 00.

24 Set random record field. On entry, DS:DX point to an opened FCB. This function sets the random record field to the same file address as the current block and record fields.

25 Set vector. The interrupt type specified in AL is set to the 4-byte address DS:DX.

26 Create a new program segment. On entry, DX has a segment number at which to set up a new program segment. The entire 100 hex area at location zero in the current program segment is copied into location zero in the new program segment. The memory size information at location 6 is updated and the current termination and <CONTROL-C> exit addresses are saved in the new program segment starting at 0A hex

27 Random block read. On entry, DS:DX point to an opened FCB, and CX contains a record count that must not be zero. The specified number of records (in terms of the record size field) are read from the file address specified by the random record field

into the disk transfer address. If end-of-file is reached before all records have been read, AL returns either 01 or 03. A return of 01 indicates end-of-file and the last record is complete, a 03 indicates the last record is a partial record. If wrap-around above address FFFF hex in the disk transfer segment would occur, as many records as possible are read and AL returns 02. If all records are read successfully, AL returns 00. In any case CX returns with the actual number of records read, and the random record field and the current block/record fields are set to address the next record.

28 Random block write. Essentially the same as function 27 above, except for writing and a write-protect indication. If there is insufficient space on the disk, AL returns 01 and no records are written. If CX is zero upon entry, no records are written, but the file is set to the length specified by the Random Record field, whether longer or shorter than the current file size (allocation units are released or allocated as appropriate).

29 Parse file name. On entry DS:SI points to a command line to parse, and ES:DI points to a portion of memory to be filled in with an unopened FCB. Leading TABs and spaces are ignored when scanning. If bit 0 of AL is equal to 1 on entry, then at most one leading file name separator will be ignored, along with any trailing TABs and spaces. The four filename separators are:

; , = +

If bit 0 of AL is equal to 1, then all parsing stops if a separator is encountered. The command line is parsed for a file name of the form d:filename.ext, and if found, a corresponding unopened FCB is created at ES:DI. The entry value of AL bits 1, 2, and 3 determine what to do if the drive, filename, or extension, respectively, are missing. In each care, if the bit is a zero and the field is not present on the command line, then the FCB is filled with a fixed value (0, meaning the default drive for the drive field ; all blanks for the filename and extension fields). If the bit is a 1, and the field is not present on the command line, then that field in the destination FCB at ES:DI is left unchanged. If an asterisk "*" appears in the filename or extension, then all remaining characters in the name or extension are set to "?".

The following characters are illegal within MS-DOS file specifications:

" / [ ] + = ; ,

Control characters and spaces also may not be given as elements of file specifications. If any of these characters are encountered while parsing, or the period (.) or colon (:) is found in an invalid position, then parsing stops at that point.

If either "?" or "*" appears in the file name or extension, then AL returns 01, otherwise 00. DS:SI will return pointing to the first character after the file name.

2A Get date. Returns date in CX:DX. CX has the year, DH has the month (1=Jan, 2=Feb, etc.), and DL has the day. If the time-of-day clock rolls over to the next day, the date will be adjusted accordingly, taking into account the number of days in each month and leap years.

2B Set date. On entry CX:DX must have a valid date in the same format as returned by function 2A above. If the date is indeed valid and the set operation is successful, then AL returns 00. If the date is not valid, then AL returns FF.

2C Get time. Returns with time-of-day in CX:DX. Time is actually represented as four 8-bit binary quantities, as follows: CH has the hours (0-23), CL has minutes (0-59), DH has seconds (0-59), DL has 1/100 seconds (0-99). This format is easily converted to a printable form yet can also be calculated upon (e.g., subtracting two times).

2D Set time. On entry, CX:DX has time in the same format as returned by function 2C above. If any component of the time is not valid, the set operation is aborted and AL returns FF. If the time is valid, AL returns 00.

2E Set/Reset Verify Flag. On entry, DL must be 0 and AL has the verify flag: 0 = no verify, 1 = verfiy after write. this flag is simply passed to the I/O system on each write, so sts exact meaning is interpreted there.

APPENDIX D

Disk Errors

If a disk error occurs at any time during any command or program, MS-DOS retries the operation three times. If the operation cannot be completed successfully, MS-DOS returns an error message in the following format:

```
<type> ERROR WHILE <I/O action> ON DRIVE x
Abort,Ignore,Retry:_
```

In this message,<type> may be one of the following:

```
WRITE PROTECT
NOT READY
SEEK
DATA
SECTOR NOT FOUND
WRITE FAULT
DISK
```

The <I/O-action> may be either of the following:

```
READING
WRITING
```

The drive <d> indicates the drive in which the error has occurred.
MS-DOS waits entry of one of the following responses:

A Abort. Terminate the program requesting the disk read or write.

I Ignore. Ignore the bad sector and pretend the error did not occur.

R Retry. Repeat the operation. This response is particularly useful if the operator has corrected the error (such as with NOT READY or WRITE PROTECT).

Usually, you will want to attempt recovery by entering responses in the order:

R (to try again)
A (to terminate program and try a new disk)

One other error message might be related to faulty disk read or write:

FILE ALLOCATION TABLE BAD FOR DRIVE x

This message means that the copy in memory of one of the allocation tables has pointers to nonexistent blocks. Possibly the disk was not FORMATted before use.

INDEX

%0 dummy (Batch) 2-18
.BAT (extension) 2-18
/S (Format) 3-14
<C1> 4-4
<CM> 4-5
<CT> 4-6
<INS> 4-10
<NT> 4-12
<QI> 4-9
<REP> 4-11
<S1> 4-7
<SM> 4-8

Abort D-1
AUTOEXEC.BAT 2-19

Batch 2-18

CHKDSK 1-4
Command types 2-16
COMMAND.COM 1-4
COMMaND.COM 2-3
Commands
  CHKDSK 3-3
  COPY 3-5
  DATE 3-9
  DEL 3-10
  DIR 3-11
  EXE2BIN 3-12
  FORMAT 3-14
  PAUSE 3-15
  REM 3-16
  REN 3-17
  SYS 3-18
  TIME 3-19
  TYPE 3-20
  Wild card characters (?,*) . 2-6
Concatenation 3-5
Control Characters 2-13
CREF.EXE 1-4
Cursor 2-15

Data Error D-1
DEBUG 1-4
DEBUG Commands
  (C)ompare 5-7
  (D)ump 5-8
  (E)nter 5-10
  (Fill) 5-12
  (G)o 5-13

(H)ex 5-15
(I)nput 5-16
(L)oad 5-17
(M)ove 5-18
(N)ame 5-20
(O)utput 5-23
(Q)uit 5-24
(R)egister 5-25
(S)earch 5-28
(T)race 5-29
(U)nassemble 5-31
(W)rite 5-33
Flags 5-26
DEBUG Errors
BF - Bad flag 5-35
BP - Too many breakpoints . 5-35
BR - Bad register 5-35
DF - Double flag 5-35
Default drive 2-15
Device independent I/O 2-4
Directory 2-3
Disk Error D-1
Disk Errors
Abort D-1
Data Error D-1
Disk Error D-1
File Allocation Table Bad For Drive x D-2
Ignore D-1
Not Ready Error D-1
Retry D-1
Sector Not Found Error . . . D-1
Seek Error D-1
Write Fault Error D-1
Write Protect Error D-1
Drive designations 2-4
Dummy parameters (Batch) . . . 2-18

EDLIN 1-4
EDLIN Commands
Append Lines 4-18
Delete Lines 4-19
Edit Line 4-16
End Editing 4-21
Insert Text 4-22
List Text 4-25
Quit (EDLIN) 4-27
Replace Text 4-29
Search Text 4-32
Write Lines 4-34
EDLIN Errors
Cannot edit .BAK file--rename file 4-35
Disk Full 4-36
Entry error 4-36
Line too long 4-36
No room in directory for file 4-35

EXE files 5-18
EXE2BIN 2-17
External commands 2-14, 2-17

FILCOM.COM 1-5
File allocation table 2-3
File Allocation Table Bad For Drive x D-2
File system 2-4
FORMAT . . . . . 1-4 to 1-5

Hidden files 2-1

Ignore D-1
Internal commands 2-17
Intraline Commands
  Copy multiple characters . . 4-5
  Copy one character 4-4
  Copy template 4-6
  Enter insert mode 4-10
  Exit insert mode 4-11
  New template 4-12
  Quit 4-9
  Quit input 4-9
  Replace mode 4-11
  Skip multiple characters . . 4-8
  Skip one character 4-7
IO.SYS 1-5, 2-3

LIB.EXE 1-5
LINK.EXE 1-5

MASM.EXE 1-5
Memory 2-3
MSDOS.SYS 1-6

Not Ready Error D-1

Positional parameters 2-18
Prompt 2-15
Provided software
  CHKDSK.COM 1-4
  COMMAND.COM 1-4
  CREF.EXE 1-4
  DEBUG.COM 1-4
  EDLIN.COM 1-4
  FILCOM.COM 1-5
  fORMAT.COM . . . . . 1-4 to 1-5
  IO.SYS 1-5
  LIB.EXE 1-5
  LINK.EXE 1-5
  MASM.EXE 1-5
  MSDOS.SYS 1-6
  SYS.COM 1-6

RENAME (synonym for REN) . . . 3-17

Reserved sectors 2-3, 3-14
Retry D-1

Sector Not Found Error D-1
Seek Error D-1
Start-up 1-6
Syntax Notation 1-7
SYS.COM 1-6

Template 2-8

Version number 1-6

Wild card characters 2-6
Write Fault Error D-1
Write Protect Error D-1

Microsoft®
BASIC Interpreter

for the MS™-DOS Operating System

User's Guide

Microsoft Corporation

Information in this document is subject to change without notice and does not represent a commitment on the part of Microsoft Corporation. The software described in this document is furnished under a license agreement or nondisclosure agreement. The software may be used or copied only in accordance with the terms of the agreement. It is against the law to copy Microsoft BASIC on magnetic tape, disk, or any other medium for any purpose other than the purchaser's personal use.

© Microsoft Corporation, 1981, 1982, 1983

If you have comments about the software or these manuals, please complete the Software Problem Report at the back of the *Microsoft BASIC Reference Manual* and return it to Microsoft Corporation.

Microsoft is a registered trademark of Microsoft Corporation.

MS is a trademark of Microsoft Corporation.

The Microsoft logo is a trademark of Microsoft Corporation.

Document no. 8108L-527-01
Catalog no. 1414
Part no. 14F14UG

Package Contents

Your Microsoft® BASIC Interpreter package includes:

1 floppy disk containing the following files:

BASIC86.COM, which contains MS™-BASIC Interpreter.

RANDOM.BAS, which tests random I/O logic of MS™-BASIC.

1 binder containing the following documentation:

Microsoft BASIC Reference Manual, which contains descriptions of all the features of Microsoft® BASIC. This information applies to all implementations of Microsoft BASIC.

Microsoft BASIC Interpreter User's Guide, which contains all the information that applies to a particular implementation or operating system. The *User's Guide* also contains information about converting programs to Microsoft BASIC, using disk files, and calling assembly language subroutines.

Contents

Introduction 1

Major Features of Microsoft BASIC 1
System Requirements 2
How to Use This Manual 3
Syntax Notation 4
Learning More about BASIC 5

1 Microsoft BASIC With MS-DOS 7

1.1 How to Start-up Microsoft BASIC 9
1.2 How to Exit Microsoft BASIC 11
1.3 Disk Files 12

2 Microsoft BASIC Language Differences for MS-DOS 13

2.1 BLOAD Statement 15
2.2 BSAVE Statement 17
2.3 CALL Statement 18
2.4 CALLS Statement 19
2.5 CLOAD Statement 19
2.6 COMMON Statement 20
2.7 CSAVE Statement 20
2.8 DATE$ Function 20
2.9 DATE$ Statement 21
2.10 DEF SEG Statement 22
2.11 EOF Function 23
2.12 FILES Statement 25
2.13 INP Function 25
2.14 LOF Function 26
2.15 OPEN Statement 27
2.16 OUT Statement 28
2.17 RESET Statement 29
2.18 TIME$ Function 29
2.19 TIME$ Statement 30
2.20 USR Function 30

3 Converting Programs to Microsoft BASIC 33

3.1 String Dimensions 35
3.2 Multiple Assignments 36
3.3 Multiple Statements 36
3.4 MAT Functions 36

4 Disk File Handling 37

4.1 Program File Commands 39
4.2 Protected Files 41
4.3 Disk Data Files—Sequential and Random Access I/O 41

5 Microsoft BASIC Assembly Language Subroutines 51

5.1 Memory Allocation 53
5.2 CALL Statement 54
5.3 CALLS Statement 59
5.4 USR Function 59

Appendix: MS-DOS Error Messages 63

Index 65

Introduction

Microsoft® BASIC (MS™-BASIC), Release 5, is the most extensive implementation of BASIC available for microprocessors. It meets the requirements for the ANSI subset standard for BASIC, and supports many unique features rarely found in other BASICs. In addition, Microsoft BASIC has sophisticated string handling and structured programming features that are especially suited for applications development. Microsoft BASIC language is compatible with Microsoft® BASIC Compiler. MS-BASIC gives users what they want from a BASIC—ease of use plus the features that make a micro perform like a minicomputer or large mainframe.

In 1975, Microsoft Corporation wrote the first BASIC interpreter for the microcomputer. Today MS-BASIC is the BASIC you'll find on all the largest-selling microcomputers and in the application programs sold by many manufacturers and software houses. With over a million installations, MS-BASIC is recognized as the industry standard.

Major Features of Microsoft BASIC

1. Four variable types: integer (-32768 through +32767), string (up to 255 characters), single precision floating-point (7 digits), double precision floating-point (16 digits).
2. Trace facilities (TRON/TROFF) for easier debugging.
3. Error trapping using the ON ERROR GOTO statement.
4. PEEK and POKE statements to read and write any memory location.
5. Automatic line number generation and renumbering, including referenced line numbers.
6. Arrays with multiple dimensions.
7. Boolean operators OR, AND, NOT, XOR, EQV, IMP.

8. Formatted output using the complete PRINT USING facility, including asterisk fill, floating dollar sign, scientific notation, trailing sign, and comma insertion.
9. Direct access to 64K I/O ports with the INP and OUT functions.
10. Extensive program editing facilities via EDIT command and edit mode subcommands.
11. Assembly language subroutine calls: up to 10 per program with the USR function; limited only by available memory space with the CALL statement.
12. IF/THEN/ELSE and nested IF/THEN/ELSE constructs.
13. Variable length random access and sequential access disk files with a complete set of file manipulation statements: OPEN, CLOSE, GET, PUT, KILL, NAME, MERGE.

Note that features vary from one implementation of Microsoft BASIC to the next. See Chapters 1 and 2 of this manual for an exact description of the features in your implementation of Microsoft BASIC.

System Requirements

The minimum memory requirement for MS-BASIC running under the MS-DOS operating system is 64K. This assumes 36K for MS-BASIC and 20K for MS-DOS, leaving 8K for user programs. The size of MS-DOS may vary, however, according to your system configuration. Consult your MS-DOS documentation to find the system requirement for your machine.

One disk drive is required.

If your system does not meet these minimum requirements, ask your computer dealer how to expand your system.

How to Use This Manual

Microsoft BASIC Interpreter User's Guide contains the information about MS-BASIC that is specific to your implementation. Also provided are separate chapters on converting previously written programs to MS-BASIC, handling disk files, and calling assembly language subroutines.

This introduction tells you about MS-BASIC and its special features, your system requirements, and the conventions used in syntax notation. It also lists additional sources of information about programming in BASIC.

Chapter 1, "Microsoft BASIC With MS-DOS," tells you how to start-up and exit MS-BASIC and explains the filenaming conventions.

Chapter 2, "Microsoft BASIC Language Differences for MS-DOS," explains the differences between the features in your implementation and the descriptions in *Microsoft BASIC Reference Manual.*

Chapter 3, "Converting Programs to Microsoft BASIC," describes the minor adjustments necessary when converting programs written in other BASICs to MS-BASIC.

Chapter 4, "Disk File Handling," explains disk file manipulation procedures. This chapter may be read as an overview or used for reference if you encounter any disk-related errors while running MS-BASIC.

Chapter 5, "Microsoft BASIC Assembly Language Subroutines," provides information about calling assembly language subroutines.

The appendix, "MS-DOS Error Messages," lists the error messages that are specific to your implementation of MS-BASIC.

Syntax Notation

The following notation is used throughout this manual in descriptions of command and statement syntax:

[]	Square brackets indicate that the enclosed entry is optional.
< >	Angle brackets indicate user-entered data. When the angle brackets enclose lowercase text, the user must type in an entry defined by the text; for example, <filename>. When the angle brackets enclose uppercase text, the user must press the key named by the text; for example, <RETURN>.
{ }	Braces indicate that the user has a choice between two or more entries. At least one of the entries enclosed in braces must be chosen unless the entries are also enclosed in square brackets.
|	Vertical bars separate choices within braces. At least one of the entries separated by bars must be chosen unless the entries are also enclosed in square brackets.
...	Ellipses indicate that an entry may be repeated as many times as needed or desired.
CAPS	Capital letters indicate portions of statements or commands that must be entered exactly as shown.

All other punctuation, such as commas, colons, slash marks, and equal signs, must be entered exactly as shown.

Learning More About BASIC

The manuals in this package provide complete reference information for your implementation of Microsoft BASIC. They do not, however, contain tutorial information on how to write programs in BASIC. If you are new to BASIC or need help in learning to program, we suggest you read one of the following books:

Albrecht, Robert L., Finkel, LeRoy, and Brown, Jerry. *BASIC.* New York: Wiley Interscience, 2nd ed., 1978.

Billings, Karen and Moursund, David. *Are You Computer Literate?* Beaverton, Oregon: Dilithium Press, 1979.

Coan, James. *Basic BASIC.* Rochelle Park, N. J.: Hayden Book Company, 1978.

Dwyer, Thomas A. and Critchfield, Margot. *BASIC and the Personal Computer.* Reading, Mass.: Addison Wesley Publishing Company, 1978.

Simon, David E. *BASIC From the Ground Up.* Rochelle Park, N. J.: Hayden Book Company, 1978.

Chapter 1

Microsoft BASIC With MS-DOS

1.1 How to Start-up Microsoft BASIC 9
1.2 How to Exit Microsoft BASIC 11
1.3 Disk Files 12

The MS™-DOS version of Microsoft BASIC is supplied on a standard size 3740 single density floppy disk. The name of the file is BASIC86.COM. A 68K or larger MS-DOS is recommended.

Important

Before you use MS-BASIC, make backup copies of your MS-BASIC disk(s) using the MS-DOS DISKCOPY command. (See Chapter 5, "MS-DOS Commands," in the *MS-DOS User's Guide* for more information about the DISKCOPY command.) Store your master disk(s) in a safe place and work with the backup copies.

1.1 How to Start-up Microsoft BASIC

To start-up MS-BASIC, bring up MS-DOS and enter the following command:

```
BASIC86
```

The system will reply:

```
Microsoft BASIC Version 5.xx
[MS-DOS Version]
Copyright 1977-1982 (C) by Microsoft
Created: dd-mmm-yy
xxxx Bytes Free
Ok
```

You can also specify a set of options in the command line. The syntax is:

```
BASIC86 [<filename>][/F:<number of files>]
    [/M:<highest memory location>][/S:<maximum record size>]
```

If <filename> is present, Microsoft BASIC proceeds as if a RUN <filename> command were typed after start-up is complete. A default extension of .BAS is used if none is supplied and the filename is less than nine characters long.

The <filename> option allows MS-BASIC programs to be executed in batch mode using the batch facility in MS-DOS. Such programs should include a SYSTEM statement in order to return to MS-DOS when they have finished, allowing the next program in the batch stream to be executed. See Chapters 4 and 5 of the *MS-DOS User's Guide* for more information about using the batch facility.

/F:<number of files> sets the number of disk data files that may be open at any one time during the execution of an MS-BASIC program. Each file data block allocated in this fashion requires 180 bytes of memory, plus the number of bytes set with /S: (the default size for /S: is 128 bytes). If the /F: option is omitted, the number of files defaults to 3.

/M:<highest memory location> sets the highest memory location that will be used by MS-BASIC.

/S:<maximum record size> sets the maximum record size for use with random files. The default record size is 128 bytes.

Note

<number of files>, <highest memory location>, and <maximum record size> can be in decimal, octal (preceded by &O) or hexadecimal (preceded by &H).

Examples of start-up command lines:

BASIC86 PAYROLL.BAS

Use all memory and 3 files, load and execute PAYROLL.BAS.

BASIC86 INVENT/F:6

Use all memory and 6 files, load and execute INVENT.BAS.

BASIC86 /M:32768

Use first 32K of memory and 3 files.

BASIC86 DATACK/F:2/M:&H9000

Use first 36K of memory, 2 files, and execute DATACK.BAS.

1.2 How to Exit Microsoft BASIC

To exit MS-BASIC and return to MS-DOS level, enter the command

```
SYSTEM
```

which closes all files and reloads MS-DOS into memory without deleting any existing programs or data (i.e., performs a "warm start"). <Control-C>, when executed with MS-BASIC, returns to the MS-BASIC command level, not to MS-DOS.

1.3 Disk Files

Disk filenames follow the normal MS-DOS naming conventions. All filenames may begin with a drive specification of <letter>: (e.g., A:). If no drive is specified, the current drive is assumed. The default extension .BAS is used on LOAD, SAVE, MERGE and RUN <filename> commands if no period (.) appears in the filename and the filename is less than nine characters long.

Examples

```
RUN "NEWFILE.EXE"
RUN "A:NEWFILE.EXE"
SAVE  "NEWFILE"          (File is SAVEd with .BAS extension)
```

Large random access files are supported. The maximum logical record number is 32767. If a record size of 256 is specified, files up to eight megabytes may be accessed.

Chapter 2

Microsoft BASIC Language Differences for MS-DOS

2.1 BLOAD Statement 15
2.2 BSAVE Statement 17
2.3 CALL Statement 18
2.4 CALLS Statement 19
2.5 CLOAD Statement 19
2.6 COMMON Statement 20
2.7 CSAVE Statement 20
2.8 DATE$ Function 20
2.9 DATE$ Statement 21
2.10 DEF SEG Statement 22
2.11 EOF Function 23
2.12 FILES Statement 25
2.13 INP Function 25
2.14 LOF Function 26
2.15 OPEN Statement 27
2.16 OUT Statement 28
2.17 RESET Statement 29
2.18 TIME$ Function 29
2.19 TIME$ Statement 30
2.20 USR Function 30

The following features differ in this implementation from the descriptions given in the *Microsoft BASIC Reference Manual.*

2.1 BLOAD Statement

The BLOAD statement allows a program or data that has been saved as a memory image file to be loaded anywhere in memory. A memory image file is a byte-for-byte copy of what was originally in memory. See Section 2.2, "BSAVE Statement," for information about saving memory image files.

BLOAD is often used for, but is not restricted to, loading assembly language programs. It may also be used to load compiled Microsoft® Pascal or Microsoft® FORTRAN routines, for example.

Syntax BLOAD <filename>[,<offset>]

<filename> is a string expression containing the device designation and filename. (The device designation is optional).

<offset> is a numeric expression returning an unsigned integer in the range 0 to 65535. This is the offset address at which loading is to start in the segment declared by the last DEF SEG statement (see Section 2.10, "DEF SEG Statement").

Purpose To load the specified memory image file into memory.

Remarks BLOAD observes the following rules:

1. If the device is omitted, the current drive is assumed.
2. If the offset is omitted, the segment address and offset contained in the file (i.e., the address specified by the BSAVE statement when the file was created) are used. Therefore, the file is loaded into the same location from which it was saved.

3. If the offset is specified, the segment address used is the one given in the most recently executed DEF SEG statement. Therefore, a DEF SEG statement should be executed before the BLOAD statement.

Warning

BLOAD does not perform an address range check. It is therefore possible to load a file anywhere in memory. The user must be careful not to load over MS-BASIC or the operating system.

Example

```
10 'Load subroutine at 6000:F000
20 DEF SEG = &H6000 'Set segment to 6000 Hex
30 BLOAD"PROG1",&HF000 'Load PROG1
```

This example sets the segment address at 6000 Hex and loads PROG1 at F000.

Not all versions of MS-BASIC support the BLOAD statement. Therefore it is not discussed in the *Microsoft BASIC Reference Manual.*

2.2 BSAVE Statement

The BSAVE statement allows data or programs to be saved as memory image files on disk. A memory image file is a byte-for-byte copy of what is in memory. BSAVE is often used for saving assembly language programs, but it can also be used with data or programs written in other languages. The BSAVE statement is not discussed in the *Microsoft BASIC Reference Manual.*

Syntax

BSAVE<filename>,<offset>,<length>

<filename> is a string expression containing the device designation and filename. (The device designation is optional).

<offset> is a numeric expression returning an unsigned integer in the range 0 to 65535. This is the offset address to start saving from in the segment declared by the most recently executed DEF SEG statement.

<length> is a numeric expression returning an unsigned integer in the range 1 to 65535. This is the length in bytes of the memory image file to be saved.

Purpose

To save the contents of the specified area of memory as a disk file.

Remarks

The <filename>, <offset>, and <length> are required in the syntax.

A DEF SEG statement should be executed before the BSAVE statement. The address given in the DEF SEG statement will be used for the save.

Example

```
10 'Save PROG1
20 DEF SEG = &H6000
30 BSAVE"PROG1",&HF000,256
```

This example saves 256 bytes starting at 6000:F000 in the file PROG1.

2.3 CALL Statement

The CALL statement is used to interface machine language programs with MS-BASIC. The following discussion supplements the one presented in the *Microsoft BASIC Reference Manual.*

Syntax

CALL <variable name> [(<argument list>)]

<variable name> contains the segment offset that is the starting point in memory of the subroutine being called. Note that the segment offset must be assigned to the variable before the CALL statement is issued (see following example).

<argument list> contains the variables or constants, separated by commas, that are to be passed to the routine.

If this list contains an array variable followed by an undefined simple variable, an "Illegal Function Call" error will result. This error can be avoided by assigning a value to all simple variables prior to executing the CALL statement.

Purpose

To call an assembly language subroutine.

Remarks

The CALL statement is the recommended way of calling 8086 machine language programs with Microsoft BASIC. We suggest you do not use the USR function. See Chapter 5, "Microsoft BASIC Assembly Language Subroutines," for comparison of the two methods and for a complete description of using the CALL statement for assembly language subroutines.

When CALL is executed, control is transferred to the user's routine via the segment address given in the last DEF SEG statement and the segment offset specified by the <variable name> portion of the CALL statement. Values are returned to MS-BASIC by including the variable name which will receive the result in the <argument list>.

Example

```
100 DEF SEG = &H8000
110 FOO = &H7FA
120 CALL FOO(A,B$,C)
 .
 .
 .
```

Line 100 sets the segment address to 8000 Hex. The variable FOO is set to &H7FA, so that the call to FOO will execute the subroutine at location 8000:7FA Hex (equivalent to absolute address 807FA Hex).

2.4 CALLS Statement

The CALLS statement is just like CALL, except the segmented addresses of all arguments are passed. (CALL passes unsegmented addresses). CALLS should be used when accessing MS™-FORTRAN routines, since all MS-FORTRAN parameters are "call-by-reference" segmented addresses.

As with the CALL statement, CALLS uses the segment address defined by the most recently executed DEF SEG statement to locate the routine being called.

The CALLS statement is not included in the *Microsoft BASIC Reference Manual.*

2.5 CLOAD Statement

The CLOAD statement is not supported by this implementation of MS-BASIC.

2.6 COMMON Statement

As explained in Section 2.7, "COMMON," of the *Microsoft BASIC Reference Manual,* array variables are specified in a COMMON statement by appending parentheses to the variable name. Because other Microsoft products allow the number of dimensions in the array to be included in the statement, MS-BASIC Interpreter will accept that syntax but will ignore the numeric expression itself. For example, the following statements are both valid and considered equivalent:

```
COMMON A( )
COMMON A(3)
```

The number in parentheses is the number of dimensions, not the dimensions themselves. For example, the variable A(3) in this example might correspond to a DIM statement of DIM A(5,8,4).

2.7 CSAVE Statement

The CSAVE statement is not supported by this implementation of MS-BASIC.

2.8 DATE$ Function

The DATE$ function retrieves the current date. This function is not included in the *Microsoft BASIC Reference Manual.*

Syntax DATE$

Purpose To retrieve the current date. (To set the date, use the DATE$ statement, described in Section 2.9, "DATE$ Statement.")

Remarks The DATE$ function returns a ten-character string in the form mm-dd-yyyy, where mm is the month (01 through 12), dd is the day (01 through 31), and yyyy is the year (1980 through 2099).

Example

```
10 PRINT DATE$
```

The DATE$ function prints the date, calculated from the date set with the DATE$ statement.

2.9 DATE$ Statement

The DATE$ statement sets the current date. This statement is not included in the *Microsoft BASIC Reference Manual.*

Syntax

```
DATE$ = <string expression>
```

<string expression> returns a string in one of the following forms:

```
mm-dd-yy
mm-dd-yyyy
mm/dd/yy
mm/dd/yyyy
```

Purpose To set the date for the DATE$ function.

Example

```
10 DATE$ = "01-01-1981"
```

The current date is set at January 1, 1981.

2.10 DEF SEG Statement

Syntax

DEF SEG [= <address>]

<address> is a numeric expression returning an unsigned integer in the range 0 to 65535.

Purpose

To assign the current segment address to be referenced by a subsequent CALL, CALLS, or POKE statement or by a USR or PEEK function.

Remarks

The address specified is saved for use as the segment required by CALL, CALLS, POKE, USR, and PEEK.

Entry of any value outside the <address> range 0 through 65535 will result in an "Illegal Function Call" error, and the previous value will be retained.

If the <address> option is omitted, the segment to be used is set to the MS-BASIC data segment (DS). This is the initial default value.

If the <address> option is given, it should be based on a 16-byte boundary. MS-BASIC does not check the validity of the specified address.

Note

DEF and SEG must be separated by a space. Otherwise, MS-BASIC will interpret the statement DEFSEG=100 to mean "assign the value 100 to the variable DEFSEG."

Example

```
10 DEF SEG = &HB800 'Set segment to &800 Hex
20 DEF SEG 'Restore segment to MS-BASIC data
             segment.
```

The DEF SEG statement is not included in the *Microsoft BASIC Reference Manual.*

2.11 EOF Function

The EOF function may be used with random access files as well as sequential files. The following discussion supplements the one presented in the *Microsoft BASIC Reference Manual.*

Syntax EOF (<file number>)

Purpose For sequential files, the EOF function returns true (-1) if there is no more data in the file. The file is empty if the next input operation (INPUT, LINE INPUT, INPUTS, for example) would cause an "Input past end" error.

For random access files, the EOF function returns true (-1) if the most recently executed GET statement attempts to read beyond the end-of-file.

Remarks Because MS-BASIC allocates 128 bytes to a file at a time, it is possible that EOF will not accurately detect the end of a random access file which has been opened with a record length of less than 128 bytes. For example, if a file is opened with a record length of 64 bytes and one record is written to the file (e.g., PUT 1,1), EOF will return false if a GET statement is attempted on the second record of the file (e.g., GET 1,2). This will occur even though this record has not actually been written to.

Example

```
10 REM
20 REM          Open the library catalog file,
30 REM          LIBRARY.DAT.
40 OPEN"R",1,"LIBRARY.DAT"
50 REM          The first 35 bytes of the
60 REM          record contain the title,
70 REM          the remaining 93 bytes con-
80 REM          tain additional information which
90 REM          is not used by this program.
100 FIELD 1,35 AS TITLE$, 93 AS G$
110 REM
120 REM         Initialize the number of books seen.
130 REM
140 NBOOKS = 0
```

```
150 REM         Attempt to fetch the next record.
160 REM         Note that the record number
170 REM         argument of GET isn't specified
180 REM         so the next record of
190 REM         the file is fetched.
200 GET 1
210 REM
220 REM         Is this the end of file?
230 REM
240 IF EOF(1) THEN 1000
250 REM         No, increment the count of books,
260 REM         print the current title, and
270 REM         loop back to read the next
280 REM         record.
290 NBOOKS = NBOOKS + 1
300 PRINT TITLE
310 GOTO 200
1000 REM        Here when the end of file has
1010 REM        been reached. Print a blank line
1020 REM        and the number of books, close the
1030 REM        file and terminate the program.
1040 PRINT      "There are";NBOOKS;" books in";
1050 PRINT      " your library."
1060 CLOSE
1070 END
```

This sample program lists the titles of the books cataloged in the file LIBRARY.DAT. It also counts the books in the library by counting the number of records which are read from LIBRARY.DAT before the end-of-file is encountered.

Each record of LIBRARY.DAT contains information on one book in the library. The record length is 128 bytes. The first 35 bytes contain the title of the book, the remaining 93 bytes contain additional information about the book (e.g., author, publisher, location, etc.) This information is not used in this example.

2.12 FILES Statement

Syntax FILES[<filename>]

<filename> includes a filename and optional device designation.

Purpose To print the names of files residing on the specified disk.

Remarks If <filename> is omitted, all the files on the currently selected drive will be listed. <filename> is a string formula which may contain question marks (?) or asterisks (*) used as wild cards. A question mark will match any single character in the filename or extension. An asterisk will match one or more characters starting at that position. The asterisk is a shorthand notation for a series of question marks.

Examples

```
FILES
FILES "*.BAS"
FILES "B:*.*"
FILES "B:"              (equivalent to "B:*.*")
FILES "TEST?.BAS"
```

The FILES statement is not included in the *Microsoft BASIC Reference Manual.*

2.13 INP Function

Under this implementation of MS-BASIC, the range of valid port numbers is 0 through 65535, rather than the 0 through 255 range given in the *Microsoft BASIC Reference Manual.*

Syntax INP(I)

I is a valid machine port number in the range 0 through 65535.

Purpose To return the byte read from port I.

Remarks INP is the complementary function to the OUT statement.

Example

```
100 A = INP(54321)
```

In assembly language, this is equivalent to:

```
MOV DX,54321
IN  AL,DX
```

2.14 LOF Function

Because the LOF function is not included in all implementations of MS-BASIC, it is not included in the *Microsoft BASIC Reference Manual.*

Syntax LOF(<file number>)

Purpose To return the length of the file in bytes.

Example

```
110 IF REC*RECSIZ>LOF(1)
    THEN PRINT"INVALID ENTRY"
```

In this example, the variables REC and RECSIZ contain the record number and record length, respectively. The calculation determines whether the specified record is beyond the end-of-file.

2.15 OPEN Statement

For your implementation of MS-BASIC, an additional form of the OPEN statement is available. In addition to supporting sequential input, sequential output, and random access modes, the new form of the OPEN statement supports the APPEND mode, which allows data to be added to the end of a sequential file. The new syntax is explained below. The syntax given in the *Microsoft BASIC Reference Manual* also remains valid.

Syntax

OPEN <filename>[FOR<mode>]
AS [#]<file number> [LEN = <record length>]

<mode> can be one of the following:

OUTPUT	Specifies sequential output mode.
INPUT	Specifies sequential input mode.
APPEND	Specifies sequential output mode and sets the file pointer at the end of the file and the record number as the last record of the file. Either a PRINT# or WRITE# statement will then extend the file.

<record length> sets the record length for random access files. Do not use this option with sequential files.

The record length cannot exceed the maximum set with /S: at start-up (the default for /S: is 128 bytes). If you do not use the record length option, the default length is 128 bytes.

Remarks

If <mode> is omitted, the default random access mode is assumed.

Example 10 OPEN "MAILING.DAT" FOR APPEND AS 1

Data may now be added to the end of the file MAILING.DAT.

All other parameters and rules apply as explained in Section 2.45, "OPEN," of the *Microsoft BASIC Reference Manual.*

2.16 OUT Statement

Under this implementation of MS-BASIC, the range of valid entries for port is 0 through 65535, rather than the 0 through 255 range given in the *Microsoft BASIC Reference Manual.* The range for data is still 0 through 255.

Syntax OUT I, J

I is an integer expression in the range 0 to 65535, J is an integer expression in the range 0 to 255. I is a machine port number and J is the data to be transmitted.

Purpose To send a byte to a machine output port.

Remarks OUT is the complementary statement to the INP function.

Example 100 OUT 12345,255

In assembly language, this is equivalent to:

```
MOV  DX,12345
MOV  AL,255
OUT  DX,AL
```

2.17 RESET Statement

Because RESET does not work the same way for all implementations, it is not included in the *Microsoft BASIC Reference Manual.*

Syntax RESET

Purpose To close all files on all drives.

Remarks RESET closes all open files on all drives and writes the directory track to every disk with open files.

All files must be closed before a disk is removed from its drive.

2.18 TIME$ Function

Because this function is not included in all implementations, it is not included in the *Microsoft BASIC Reference Manual.*

Syntax TIME$

Purpose To retrieve the current time. (To set the time, use the TIME$ statement, described in Section 2.19.)

Remarks The TIME$ function returns an eight-character string in the form hh:mm:ss, where hh is the hour (00 through 23), mm is minutes (00 through 59), and ss is seconds (00 through 59). A 24-hour clock is used; 8:00 P.M., therefore, would be shown as 20:00:00.

Example

```
10 PRINT TIME$
```

Prints the time, calculated from the time set with the TIME$ statement.

2.19 TIME$ Statement

The TIME$ statement sets the current time. This statement is not included in the *Microsoft BASIC Reference Manual.*

Syntax TIME$ = <string expression>

<string expression> returns a string in one of the following forms:

hh	Sets the hour; minutes and seconds default to 00
hh:mm	Sets the hour and minutes; seconds default to 00
hh:mm:ss	Sets the hour, minutes, and seconds

Purpose To set the time for the TIME$ function.

Remarks A 24-hour clock is used; 8:00 P.M., therefore, would be entered as 20:00:00.

Example

```
10 TIME$ = "08:00:00"
```

The current time is set at 8:00 A.M.

2.20 USR Function

USR functions may be used to call assembly language subroutines, although we recommend that you use the CALL statement instead. This discussion is included for those of you who are working with previously-written programs that contain USR functions.

See Chapter 5, "Microsoft BASIC Assembly Language Subroutines," for comparison of CALL and USR and for a detailed discussion of calling assembly language subroutines.

USR is also discussed in the *Microsoft BASIC Reference Manual.* (See Section 3.41, "USR.") The discussion given here is more detailed, however, and contains some information specific to this implementation of MS-BASIC.

Syntax

USR[<digit>][(<argument>)]

<digit> specifies which USR function routine is being called. See the *Microsoft BASIC Reference Manual,* Section 2.13, "DEF USR," for the rules governing <digit>. If <digit> is omitted, USR0 is assumed.

<argument> is the value passed to the subroutine. It may be any numeric or string expression.

Purpose

To call an assembly language subroutine.

Remarks

In this implementation, if a segment other than the default segment (MS-BASIC data segment, DS) is to be used, a DEF SEG statement must be executed prior to a USR function call. The address given in the DEF SEG statement determines the segment address of the subroutine.

For each USR function, a corresponding DEF USR statement must be executed to define the USR call offset. This offset and the currently active DEF SEG segment address determine the starting address of the subroutine.

Example

```
100 DEF SEG = &H8000
110 DEF USR0 = 0
120 X = 5
130 Y = USR0(X)
140 PRINT Y
```

The type (numeric or string) of the variable receiving the function call must be consistent with the argument passed.

Chapter 3

Converting Programs to Microsoft BASIC

3.1 String Dimensions 35

3.2 Multiple Assignments 36

3.3 Multiple Statements 36

3.4 MAT Functions 36

If you have programs written in a BASIC other than Microsoft BASIC, some minor adjustments may be necessary before running them with Microsoft BASIC. This chapter lists some specific things to look for when converting programs to MS-BASIC.

3.1 String Dimensions

Delete all statements used to declare the length of strings. A statement such as DIM A$(I,J), which dimensions a string array for J elements of length I, should be converted to the MS-BASIC statement DIM A$(J).

Some BASICs use a comma (,) or ampersand (&) for string concatenation. Each of these must be changed to a plus sign (+), which is the operator for MS-BASIC string concatenation.

In MS-BASIC, the MID$, RIGHT$, and LEFT$ functions take substrings of strings. Forms such as A$(I) (to access the Ith character in A$) or A$(I,J) (to take a substring of A$ from position I to position J), must be changed as follows:

Other BASIC	Microsoft BASIC
X$ = A$(I)	X$ = MID$(A$,I,1)
X$ = A$(I,J)	X$ = MID$(A$,I,J-I+1)

If the substring reference is on the left side of an assignment and X$ replaces characters in A$, convert the reference as follows:

Other BASIC	Microsoft BASIC
A$(I) = X$	MID$(A$,I,1) = X$
A$(I,J) = X$	MID$(A$,I,J-I+1) = X$

3.2 Multiple Assignments

Some BASICs allow statements of the form

```
10 LET B = C = 0
```

to set B and C equal to zero. MS-BASIC interprets the second equal sign as a logical operator and sets B equal to -1 if C equals 0. Instead, convert this statement to two assignment statements:

```
10 C = 0:B = 0
```

3.3 Multiple Statements

Some BASICs use a backslash (\) to separate multiple statements on a line. With MS-BASIC, all statements on a line must be separated by a colon (:).

3.4 MAT Functions

Programs using the MAT functions available in some BASICs must be rewritten using FOR...NEXT loops to execute properly.

Chapter 4
Disk File Handling

4.1 Program File Commands 39

4.2 Protected Files 41

4.3 Disk Data Files—
Sequential and Random Access I/O 41

4.3.1 Sequential Files 41

4.3.1.1 Creating a Sequential File 42

4.3.1.2 Accessing a Sequential File 43

4.3.1.3 Adding Data to a Sequential File 44

4.3.2 Random Access Files 44

4.3.2.1 Creating a Random Access File 45

4.3.2.2 Accessing a Random Access File 46

Disk input and output procedures for the beginning BASIC user are examined in this chapter. If you are new to BASIC or if you're getting disk-related errors, read through these procedures and program examples to make sure you're using all the disk statements correctly.

Wherever a filename is required in a disk command or statement, use a name that conforms to the filenaming conventions for your operating system. MS-DOS will append a default extension .BAS to the filename given in a SAVE, RUN, MERGE or LOAD command.

4.1 Program File Commands

The following list reviews the commands and statements used in program file manipulation.

Note

As of MS-BASIC Release 5.27, the asterisk (*) and question mark (?) can be used as wild cards with the FILES, KILL, and NAME commands.

Use wild cards with care. Remember, for example, that the command KILL *.* will delete all files in the directory, not just the one named *.*. Chapter 3, "More About Files," of the *MS-DOS User's Guide* explains how to use wild cards.

SAVE<filename>[,A]

Writes to disk the program that currently resides in memory. Optional A writes the program as a series of ASCII characters. (Otherwise, Microsoft BASIC uses a compressed binary format.)

(See Section 4.2, "Protected Files," for discussion on how to save protected files.)

LOAD <filename>[,R]

Loads the program from disk into memory. Optional R runs the program immediately. LOAD always deletes the current contents of memory and closes all files before loading. If R is included, however, open data files are kept open. Thus, programs may be chained or loaded in sections and access the same data files. (LOAD <filename>,R and RUN <filename>,R are equivalent.)

RUN <filename>[,R]

RUN <filename> loads the program from disk into memory and runs it. RUN deletes the current contents of memory and closes all files before loading the program. If the R option is included, however, all open data files are kept open. (RUN <filename>,R and LOAD <filename>,R are equivalent.)

MERGE <filename>

Loads the program from disk into memory but does not delete the current contents of memory. The program line numbers on disk merge with the line numbers in memory. If two lines have the same number, only the line from the disk program is saved. After a MERGE command, the "merged" program resides in memory, and MS-BASIC returns to command level.

KILL <filename>

Deletes the file from the disk. <filename> may be a program file, or a sequential or random access data file.

NAME <old filename> AS <new filename>

To change the name of a disk file, execute the NAME statement, NAME <old filename> AS <new filename>. NAME may be used with program files, random access files, or sequential files.

4.2 Protected Files

If you wish to save a program in an encoded binary format, use the protect option with the SAVE command. For example:

```
SAVE "MYPROG",P
```

Warning

Because a program saved in this manner cannot be listed or edited, you may want to save an unprotected copy of the program for listing and editing.

4.3 Disk Data Files— Sequential and Random Access I/O

There are two types of disk data files that may be created and accessed by a BASIC program: sequential files and random access files.

4.3.1 Sequential Files

Sequential files are easier to create than random access files but limit flexibility and speed when it comes to accessing the data. The data written to a sequential file are ASCII characters. These characters are stored, one item after another (sequentially), in the order they are sent. They are read back the same way.

The statements and functions used with sequential files are:

CLOSE	LINE INPUT#	PRINT#
EOF	LOF	PRINT# USING
INPUT#	OPEN	WRITE#

See the *Microsoft BASIC Reference Manual* for more information on these statements and functions. Refer to Section 2.14 of this manual, "LOF Function," for more information on the LOF function.

4.3.1.1 Creating a Sequential File

The following program steps are required to create a sequential file and access the data in the file:

1. OPEN the file in "O" mode.

```
OPEN "O", #1, "DATA"
```

2. Write data to the file using the WRITE# statement. (You may use the PRINT# command instead, but consult Section 2.51 in the *Microsoft BASIC Reference Manual* before doing so.)

```
WRITE#1,A$;B$;C$
```

3. To access the data in the file, you must CLOSE the file and reOPEN it in "I" mode.

```
CLOSE #1
OPEN "I",#1,"DATA"
```

4. Use the INPUT# statement to read data from the sequential file into the program.

```
INPUT#1,X$,Y$,Z$
```

A program that creates a sequential file can also write formatted data to the disk with the PRINT# USING statement. For example, the statement

```
PRINT#1,USING"####.##,";A,B,C,D
```

could be used to write numeric data to disk without explicit delimiters. The comma (,) at the end of the format string serves to separate the items in the disk file.

The LOC function, when used with a sequential file, returns the number of sectors that have been written to or read from the file since it was opened.

Program 1 is a short program that creates a sequential file, named "DATA," from information you input at the terminal.

Program 1—Create a Sequential Data File

```
10 OPEN "O",#1,"DATA"
20 INPUT "NAME";N$
25 IF N$ = "DONE" THEN END
30 INPUT "DEPARTMENT";D$
40 INPUT "DATE HIRED";H$
50 PRINT#1,N$;",";D$;",";H$
60 PRINT: GOTO 20
RUN
NAME? MICKEY MOUSE
DEPARTMENT? AUDIO/VISUAL AIDS
DATE HIRED? 01/12/72

NAME? SHERLOCK HOLMES
DEPARTMENT? RESEARCH
DATE HIRED? 12/03/65

NAME? EBENEEZER SCROOGE
DEPARTMENT? ACCOUNTING
DATE HIRED? 04/27/78

NAME? SUPER MANN
DEPARTMENT? MAINTENANCE
DATE HIRED? 08/16/78

NAME? etc.
```

4.3.1.2 Accessing a Sequential File

Program 2 accesses the file "DATA" that was created in Program 1 and displays the name of everyone hired in 1978.

Program 2—Access a Sequential File

```
10 OPEN "I", #1 ,"DATA"
20 INPUT#1,N$,D$,H$
30 IF RIGHT$(H$,2) = "78" THEN PRINT N$
40 GOTO 20
RUN
EBENEEZER SCROOGE
SUPER MANN
Input past end in 20
Ok
```

Program 2 reads, sequentially, every item in the file. When all the data has been read, line 20 causes an "Input past end" error. To avoid getting this error, insert line 15, which uses the EOF function to test for end-of-file.

```
15 IF EOF(1) THEN END
```

Then change line 40 to GOTO 15.

4.3.1.3 Adding Data to a Sequential File

If you have a sequential file on disk and want to add more data to the end of it, you cannot simply open the file in "O" mode and start writing data. As soon as you open a sequential file in "O" mode, you destroy its current contents. Instead, use the OPEN statement with the APPEND mode, as described in Chapter 2, "Microsoft BASIC Language Differences for MS-DOS."

4.3.2 Random Access Files

Creating and accessing random access files requires more program steps than with sequential files, but there are advantages to using random access files. One advantage is that random access files require less room on the disk, because MS-BASIC stores them in a packed binary format. (A sequential file is stored as a series of ASCII characters.)

The biggest advantage to random access files is that data can be accessed randomly, i.e., anywhere on the disk. It is not necessary to read through all the information on disk with random access files, as with sequential files. This is possible because the information is stored and accessed in distinct units called records and each record is numbered.

The statements and functions that are used with random access files are:

CLOSE	GET	MKS$
CVD	LSET	OPEN
CVI	LOC	PUT
CVS	MKD$	RSET
FIELD	MKI$	

4.3.2.1 Creating a Random Access File

Creation of a random access file requires the following program steps.

1. OPEN the file for random access ("R" mode). This example specifies a record length of 32 bytes. If the record length is omitted, the default is 128 bytes.

```
OPEN "R",#1, "FILE", 32
```

2. Use the FIELD statement to allocate space in the random buffer for the variables that will be written to the random access file.

```
FIELD#1, 20 AS N$,
4 AS A$, 8 AS P$
```

3. Use the LSET command to move the data into the random buffer. Numeric values must be made into strings when placed in the buffer. To do this, use the "make" functions. MKI$ makes an integer value into a string, MKS$ makes an integer value into a single precision value, and MKD$ makes an integer value into a double precision value. See the *Microsoft BASIC Reference Manual* for more information on these functions.

```
LSET N$ = X$
LSET A$ = MKS$(AMT)
P$ = TEL$
```

4. Write the data from the buffer to the disk using the PUT statement.

```
PUT#1,CODE%
```

The LOC function, with random access files, returns the "current record number." The current record number is one plus the last record number that was used in a GET or PUT statement. For example, the statement

```
IF LOC(1)>50 THEN END
```

ends program execution if the current record number in file #1 is higher than 50.

Program 3 writes information that is input at the terminal to a random access file.

Program 3—Create a Random Access File

```
10 OPEN "R",#1,"FILE",32
20 FIELD #1,20 AS N$, 4 AS A$, 8 AS P$
30 INPUT "2-DIGIT CODE";CODE%
40 INPUT "NAME";X$
50 INPUT "AMOUNT";AMT
60 INPUT "PHONE";TEL$:PRINT
70 LSET N$ = X$
80 LSET A$ = MKS$(AMT)
90 LSET P$ = TEL$
100 PUT #1,CODE%
110 GOTO 30
```

Each time the PUT statement is executed, a record is written to the file. The two-digit code that is input in line 30 becomes the record number.

Important

Do not use a FIELDed string variable in an INPUT or LET statement. This causes the pointer for that variable to point into string space instead of into the random access file buffer.

4.3.2.2 Accessing a Random Access File

Creation of a random access file requires the following steps.

1. OPEN the file in "R" mode.

```
OPEN "R",#1,"FILE",32
```

2. Use the FIELD statement to allocate space in the random buffer for the variables that will be read from the file.

```
FIELD #1 20 AS N$,
4 AS A$, 8 AS P$
```

Note

In a program that performs both input and output on the same random access file, you can often use just one OPEN statement and one FIELD statement.

---

3. Use the GET statement to move the desired record into the random buffer.

```
GET #1,CODE%
```

4. The data in the buffer may now be accessed by the program. Numeric values must be converted back to numbers using the "convert" functions. CVI converts numeric values to integer values, CVS converts numeric values to single precision values, and CVD converts numeric values to double precision values.

```
PRINT N$
PRINT CVS(A$)
```

Program 4 accesses the random access file "FILE" that was created in Program 3. When the three-digit code is entered at the terminal, the information associated with that code is read from the file and displayed.

Program 4—Access a Random Access File

```
10 OPEN "R",#1,"FILE",32
20 FIELD #1, 20 AS N$, 4 AS A$, 8 AS P$
30 INPUT "2-DIGIT CODE";CODE%
40 GET #1, CODE%
50 PRINT N$
60 PRINT USING "$$###.##";CVS(A$)
70 PRINT P$:PRINT
80 GOTO 30
```

Program 5 is an inventory program that illustrates random file access. In this program, the record number is used as the part number, and it is assumed the inventory will contain no more than 100 different part numbers. Lines 900 through 960 initialize the data file by writing CHR$(255) as the first character of each record. This is used later (line 270 and line 500) to determine whether an entry already exists for that part number.

Lines 140 through 210 display the different inventory functions that the program performs. When you type in the desired function number, line 230 branches to the appropriate subroutine.

Program 5—Inventory

```
120 OPEN"R",#1,"INVEN.DAT",39
130 FIELD#1,1 AS F$,30 AS D$,2 AS Q$,2 AS R$,4 AS P$
140 PRINT:PRINT "FUNCTIONS:":PRINT
150 PRINT 1,"INITIALIZE FILE"
160 PRINT 2,"CREATE A NEW ENTRY"
170 PRINT 3,"DISPLAY INVENTORY FOR ONE PART"
180 PRINT 4,"ADD TO STOCK"
190 PRINT 5,"SUBTRACT FROM STOCK"
200 PRINT 6,"DISPLAY ALL ITEMS BELOW REORDER LEVEL"
210 PRINT:PRINT:INPUT"FUNCTION";FUNCTION
220 IF (FUNCTION<1)OR(FUNCTION>6)THEN PRINT
    "BAD FUNCTION NUMBER":GOTO 140
230 ON FUNCTION GOSUB 900, 250, 390, 480, 560, 680
240 GOTO 210
250 REM BUILD NEW ENTRY
260 GOSUB 840
270 IF ASC(F$)<>255 THEN INPUT"OVERWRITE";A$:
    IF A$<>"Y" THEN RETURN
280 LSET F$ = CHR$(0)
290 INPUT "DESCRIPTION";DESC$
300 LSET D$ = DESC$
310 INPUT "QUANTITY IN STOCK";Q%
320 LSET Q$ = MKI$(Q%)
330 INPUT "REORDER LEVEL";R%
340 LSET R$ = MKI$(R%)
350 INPUT "UNIT PRICE";P
360 LSET P$ = MKS$(P)
370 PUT#1,PART%
380 RETURN
390 REM DISPLAY ENTRY
400 GOSUB 840
410 IF ASC(F$) = 255 THEN PRINT "NULL ENTRY":RETURN
420 PRINT USING "PART NUMBER ###";PART%
430 PRINT D$
```

```
440 PRINT USING "QUANTITY ON HAND #####";CVI(Q$)
450 PRINT USING "REORDER LEVEL #####";CVI(R$)
460 PRINT USING "UNIT PRICE $$###.##";CVS(P$)
470 RETURN
480 REM ADD TO STOCK
490 GOSUB 840
500 IF ASC(F$)=255 THEN PRINT "NULL ENTRY":RETURN
510 PRINT D$:INPUT "QUANTITY TO ADD" ;A%
520 Q%=CVI(Q$)+A%
530 LSET Q$=MKI$(Q%)
540 PUT#1,PART%
550 RETURN
560 REM REMOVE FROM STOCK
570 GOSUB 840
580 IF ASC(F$)=255 THEN PRINT "NULL ENTRY":RETURN
590 PRINT D$
600 INPUT "QUANTITY TO SUBTRACT";S%
610 Q%=CVI(Q$)
620 IF (Q%-S%)<0 THEN PRINT "ONLY";Q%; "IN STOCK":
    GOTO 600
630 Q%=Q%-S%
640 IF Q%=<CVI(R$) THEN PRINT "QUANTITY NOW";Q%;
    " REORDER LEVEL";CVI(R$)
650 LSET Q$=MKI$(Q%)
660 PUT#1,PART%
670 RETURN
680 REM DISPLAY ITEMS BELOW REORDER LEVEL
690 FOR I=1 TO 100
710 GET#1,I
720 IF CVI(Q$)<CVI(R$) THEN PRINT D$;" QUANTITY";
    CVI(Q$) TAB(50) "REORDER LEVEL";CVI(R$)
730 NEXT I
740 RETURN
840 INPUT "PART NUMBER";PART%
850 IF(PART%<1)OR(PART%>100) THEN PRINT "BAD PART
    NUMBER": GOTO 840 ELSE GET#1,PART%:RETURN
890 END
900 REM INITIALIZE FILE
910 INPUT "ARE YOU SURE";B$;IF B$<>"Y" THEN RETURN
920 LSET F$=CHR$(255)
930 FOR I = 1 TO 100
940 PUT #1, I
950 NEXT I
960 RETURN
```

Chapter 5

Microsoft BASIC Assembly Language Subroutines

5.1 Memory Allocation 53

5.2 CALL Statement 54

5.3 CALLS Statement 59

5.4 USR Function 59

This chapter is provided for users who call assembly language subroutines from their MS-BASIC programs. If you do not use assembly language subroutines, you will not need this information.

All versions of Microsoft BASIC have provisions for interfacing with assembly language subroutines via the USR function and the CALL statement. The CALLS statement is also supported under the 8086 implementations.

The USR function allows assembly language subroutines to be called in the same way MS-BASIC intrinsic functions are called. However, CALL or CALLS statements are recommended for interfacing 8086 machine language programs with MS-BASIC. These statements produce more readable source code and can pass multiple arguments. In addition, the CALL statement is compatible with more languages than is the USR function.

5.1 Memory Allocation

Memory space must be set aside for an assembly language subroutine before it can be loaded. To do so, use the /M: switch during start-up. (See Chapter 1, "Microsoft BASIC With MS-DOS," for a review of the start-up procedure.) The /M: switch sets the highest memory location to be used by MS-BASIC.

In addition to MS-BASIC Interpreter code area, Microsoft BASIC uses up to 64K of memory beginning at its data (DS) segment.

If, when an assembly language subroutine is called, more stack space is needed, the Microsoft BASIC stack can be saved and a new stack set up for use by the assembly language subroutine. The MS-BASIC stack must be restored, however, before returning from the subroutine.

The assembly language subroutine can be loaded into memory by means of the operating system or the POKE statement. If you have the Microsoft® Utility Software Package for the 8086 Microprocessor, routines may be assembled with the Microsoft® MACRO Assembler (MS™-MACRO) and linked (but not loaded) using Microsoft® LINK Linking Loader (MS™-LINK). To load the program file, observe these guidelines:

1. Make sure the subroutines do not contain any long references.
2. Skip over the first 512 bytes of the MS-LINK output file, then read in the rest of the file.

5.2 CALL Statement

The CALL statement is the recommended way of interfacing 8086 machine language programs with MS-BASIC. Do not use the USR function (see Section 5.4, "USR Function," for a description of the function) unless you are running previously written programs that already contain USR functions.

Syntax CALL <variable name> [(<argument list>)]

<variable name> contains the segment offset that is the starting point in memory of the subroutine being called.

<argument list> contains the variables or constants, separated by commas (,), that are to be passed to the routine.

Invoking the CALL statement causes the following to occur:

1. For each parameter in the argument list, the two-byte offset of the parameter's location within the data segment (DS) is pushed onto the stack.
2. The MS-BASIC return address code segment (CS), and offset (IP) are pushed onto the stack.
3. Control is transferred to the user's routine via an 8086 long call to the segment address given in the last DEF SEG statement and the offset given in <variable name>.

Figures 1 and 2 illustrate the state of the stack at the time the CALL statement is executed, and the condition of the stack during execution of the called subroutine.

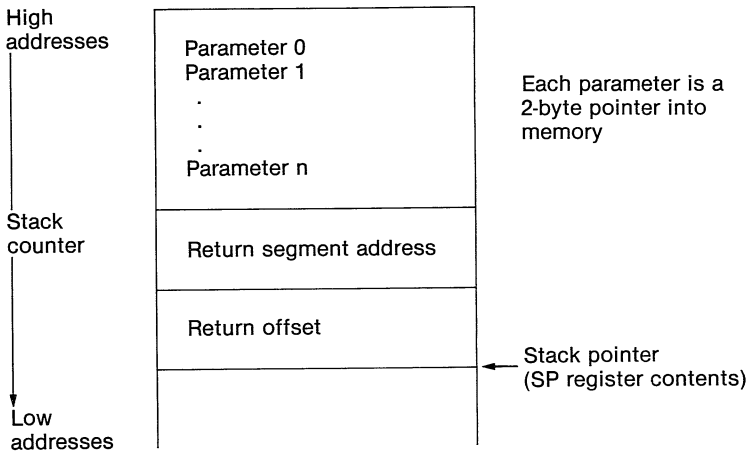


Figure 1. Stack Layout When CALL Statement Is Activated

The user's routine now has control. Parameters may be referenced by moving the stack pointer (SP) to the base pointer (BP) and adding a positive offset to BP.

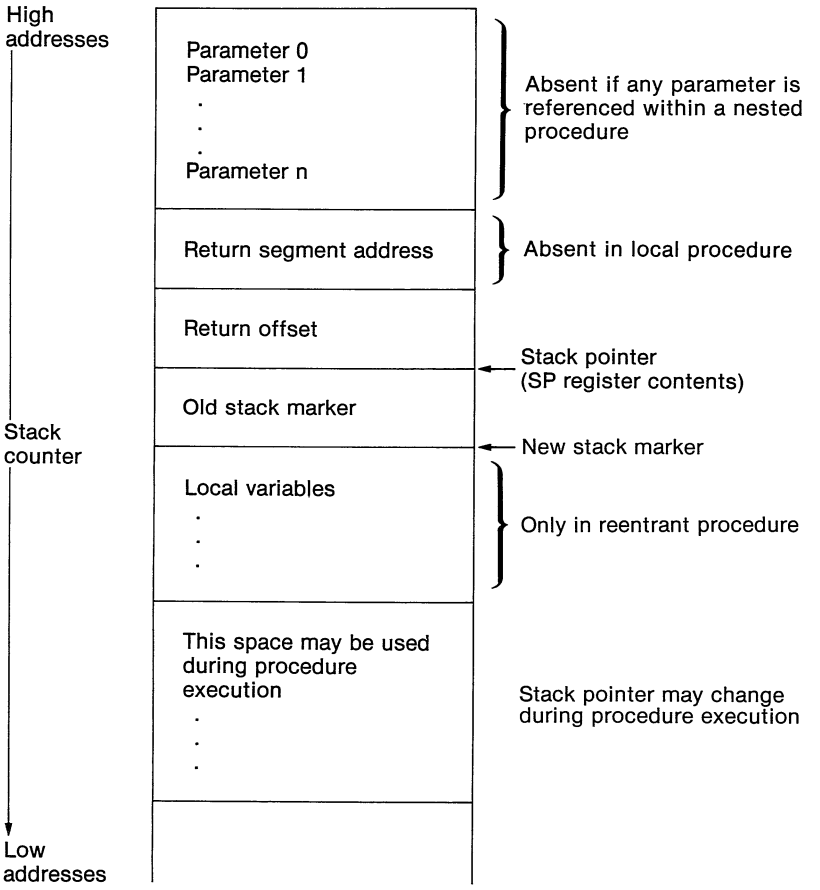


Figure 2. Stack Layout During Execution of a CALL Statement

You must observe the following rules when coding a subroutine:

1. The called routine may destroy the AX, BX, CX, DX, SI, DI, and BP registers.

2. The called program must know the number and length of the parameters passed. References to parameters are positive offsets added to BP (assuming the called routine moved the current stack pointer into BP; i.e., MOV BP,SP). That is, the location of pl is at 8(BP), p2 is at 6(BP), p3 is at 4(BP), etc.

3. The called routine must do a RET <n> statement (where <n> is two times the number of parameters in the argument list) to adjust the stack to the start of the calling sequence.

4. Values are returned to MS-BASIC by including in the argument list the variable name which will receive the result.

5. If the argument is a string, the parameter's offset points to 3 bytes which, as a unit, is called the "string descriptor." Byte 0 of the string descriptor contains the length of the string (0 to 255). Bytes 1 and 2, respectively, are the lower and upper 8 bits of the string starting address in string space.

Warning

If the argument is a string literal in the program, the string descriptor will point to program text. Be careful not to alter or destroy your program this way. To avoid unpredictable results, add + " " to the string literal in the program.

```
20 A$ = "BASIC" + "  "
```

for example, will force the string literal to be copied into string space. Now the string may be modified without affecting the program.

---

6. Strings may be altered by user routines, but the length *must not* be changed. MS-BASIC cannot correctly manipulate strings if their lengths are modified by external routines.

Example of CALL statement:

```
100 DEF SEG = &H8000
110 FOO = &H7FA
120 CALL FOO(A,B$,C)
 .
 .
 .
```

Line 100 sets the segment to 8000 Hex. The value of variable FOO is added into the address as the low word after the DEF SEG value is left shifted eight bits. (This is a function of the microprocessor, not of MS-BASIC.) Here, FOO is set to &H7FA, so that the call to FOO will execute the subroutine at location 8000:7FA Hex (absolute address 807FA Hex).

The following sequence in 8086 assembly language demonstrates access of the parameters passed. The return result is stored in the variable 'C'.

```
MOV     BP,SP          ;Get current stack posn in BP.
MOV     BX,6[BP]       ;Get address of B$ dope.
MOV     CL,[BX]        ;Get length of B$ in CL.
MOV     DX,1[BX]       ;Get addr of B$ text in DX.
          .
          .
          .
MOV     SI,8[BP]       ;Get address of 'A' in SI.
MOV     DI,4[BP]       ;Get pointer to 'C' in DI.
MOVS    WORD           ;Store variable 'A' in 'C'.
RET     6              ;Restore stack, return.
```

Important

The called program must know the variable type for the numeric parameters passed. In the previous example, the instruction

```
MOVS WORD
```

will copy only two bytes. This is fine if variables A and C are integer. You would have to copy four bytes if the variables were single precision and copy eight bytes if they were double precision.

5.3 CALLS Statement

As explained in Section 2.4, the CALLS statement should be used to access MS-FORTRAN routines. CALLS works just like CALL, except that with CALLS the arguments pass as segmented addresses, rather than as unsegmented addresses.

5.4 USR Function

Although the CALL statement is the recommended way of calling assembly language subroutines, the USR function is still available for compatibility with previously-written programs.

Syntax USR[<digit>][(<argument>)]

<digit> is from 0 to 9. <digit> specifies which USR routine is being called. If <digit> is omitted, USR0 is assumed.

<argument> is any numeric or string expression. Arguments are discussed in detail in the following paragraphs.

In this implementation of MS-BASIC, a DEF SEG statement *must* be executed prior to a USR function call to assure that the code segment points to the subroutine being called. The segment address given in the DEF SEG statement determines the starting segment of the subroutine.

For each USR function, a corresponding DEF USR statement must be executed to define the USR function call offset. This offset and the currently active DEF SEG address determine the starting address of the subroutine.

When the USR function call is made, register AL contains a value which specifies the type of argument that was given. The value in AL may be one of the following:

Value in AL	Type of Argument
2	Two-byte integer (two's complement)
3	String
4	Single precision floating-point number
8	Double precision floating-point number

If the argument is a number, the BX register pair points to the Floating-Point Accumulator (FAC) where the argument is stored:

FAC is the exponent minus 128, and the binary point is to the left of the most significant bit of the mantissa.

FAC-1 contains the highest 7 bits of mantissa with leading 1 suppressed (implied). Bit 7 is the sign of the number (0=positive, 1=negative).

If the argument is an integer:

FAC-2 contains the upper 8 bits of the argument.

FAC-3 contains the lower 8 bits of the argument.

If the argument is a single precision floating-point number:

FAC-2 contains the middle 8 bits of mantissa.

FAC-3 contains the lowest 8 bits of mantissa.

If the argument is a double precision floating-point number:

FAC-7 through FAC-4 contain four more bytes of mantissa (FAC-7 contains the lowest 8 bits).

If the argument is a string, the DX register pair points to 3 bytes which, as a unit, is called the "string descriptor." Byte 0 of the string descriptor contains the length of the string (0 to 255). Bytes 1 and 2, respectively, are the lower and upper 8 bits of the string starting address in the MS-BASIC data segment.

Warning

If the argument is a string literal in the program, the string descriptor will point to program text. Be careful not to alter or destroy your program this way.

Usually, the value returned by a USR function is the same type (integer, string, single precision, or double precision) as the argument that was passed to it.

Example

```
100 DEF USR0 = &H8000 'Assumes user gave /M:32767
120 X = 5
130 Y = USR0(X)
140 PRINT Y
```

The type (numeric or string) of the variable receiving the function call must be consistent with the argument passed.

Appendix: MS-DOS Error Messages

For this implementation of MS-BASIC, the following error messages should be added to those found in the *Microsoft BASIC Reference Manual.*

Number	Message
70	Disk write protected Your disk has a write protect tab intact, or is a disk that cannot be written to.
71	Disk not ready Could be caused by a number of problems. The most likely is that the disk is not inserted properly.
72	Disk media error A hardware or disk problem occurred while the disk was being written to or read from. For example, the disk may be damaged or the disk drive may not be working properly.
74	Rename across disks An attempt was made to rename a file with a new drive designation. This is not allowed.

Index

.BAS, 10, 39

/F:, 10
/M:, 10, 53
/S:, 10

AND, 1
ANSI standard, 1
APPEND mode, 27, 44
Assembly language subroutines, 18, 30, 51-61

BASIC86.COM, iii, 9
Bibliography of BASIC texts, 5
BLOAD, 15-16
Boolean operators, 1
BSAVE, 17

CALL, 2, 18, 22, 53-58
CALLS, 19, 22, 53, 59
CLOAD, 19
CLOSE, 2, 41, 44
COMMON, 20
Converting programs to MS-BASIC, 33-36
CSAVE, 20
CVD, 44, 47
CVI, 44, 47
CVS, 44, 47

DATE$
  function, 20-21
  statement, 21
DEF SEG, 15-17, 22, 31, 55, 58-59
DEF USR, 31, 59
Default file extension, 39
Disk file handling, 37-49

EDIT, 2
Edit mode, 2
EOF, 23-24, 41, 44
EQV, 1
Error messages, 63
Exiting MS-BASIC, 11

Features of MS-BASIC, 1
FIELD, 44, 46-47
Filenames, 12, 25, 39-40
FILES, 25, 39
Floating-Point Accumulator,(FAC), 60

GET, 2, 23, 44

IF/THEN/ELSE, 2
IMP, 1
INP, 2, 25-26
INPUT, 27, 46
INPUT#, 41

KILL, 2, 39-40

Language differences, 13-31
Learning about BASIC, 5
LEFT$, 35
LET, 46
LINE INPUT#, 41
LOAD, 39-40
LOC, 42, 44
LOF, 26, 41
LSET, 44

MAT functions, 36
Memory allocation, 53-54
Memory image file, 15-16
MERGE, 2, 39-40
Microsoft
  BASIC Compiler, 1
  BASIC Interpreter, iii, 20
  FORTRAN Compiler, 15
  FORTRAN routines, 15, 19, 59
  LINK Linking Loader, 54
  MACRO Assembler, 54
  Pascal Compiler, 15
  Pascal routines, 15
  Utility Software Package, 54
MID$, 35
MKD$, 44
MKI$, 44
MKS$, 44
MS-DOS, 7-12, 39
MS-DOS error messages, 63
MS-FORTRAN, 15, 19, 59
MS-LINK, 54
MS-MACRO Assembler, 54
Multiple assignments, 36
Multiple statements, 36

NAME, 2, 39-40
NOT, 1

ON ERROR GOTO, 1
OPEN, 2, 27-28, 41-42, 44-47
OR, 1
OUT, 2, 28
OUTPUT, 27

Package contents, iii
PEEK, 1, 22
POKE, 1, 22, 54
PRINT USING, 2
PRINT#, 41
PRINT# USING, 41-42
Program file commands, 39
Protected files, 41
PUT, 2, 44, 46

RANDOM.BAS, iii
Random access files, 41, 44-49
RET, 57
RIGHT$, 35
RSET, 29, 44
RUN, 39-40

SAVE, 39, 41
Sequential files, 41-44
Stack
  layout, 55-57
  space, 53
String
  descriptor, 57, 60-61
  dimensions, 35, 57, 60
  functions, 35
  literal, 57, 61
  space, 57
Syntax notation, 4
System requirements, 2

TIME$
  function, 29
  statement, 30
Trace facilities, 1
Trademark information, ii
TRON/TROFF, 1

USR function, 22, 30-31, 53, 59-61
USR function call, 30-31, 59-61

Warm start, 11
Wild cards, 39
WRITE#, 41

XOR, 1

Microsoft® BASIC

Reference Manual

Microsoft Corporation

Information in this document is subject to change without notice and does not represent a commitment on the part of Microsoft Corporation. The software described in this document is furnished under a license agreement or nondisclosure agreement. The software may be used or copied only in accordance with the terms of the agreement. It is against the law to copy Microsoft BASIC on magnetic tape, disk, or any other medium for any purpose other than the purchaser's personal use.

© Microsoft Corporation 1979, 1982, 1983

If you have comments about the software or this manual, please complete the Software Problem Report at the back of the *Microsoft BASIC Reference Manual* and return it to Microsoft Corporation.

Microsoft is a registered trademark of Microsoft Corporation.

MS and the Microsoft logo are trademarks of Microsoft Corporation.

Teletype is a registered trademark of Teletype Corporation.

CP/M is a registered trademark of Digital Research, Inc.

Document no. 8101-530-11
Part no. 00F14RM

Contents

Introduction 1

1 General Information about Microsoft BASIC 3

1.1 Initialization 5
1.2 Modes of Operation 5
1.3 Line Format 5
1.4 Character Set 6
1.5 Constants 8
1.6 Variables 11
1.7 Type Conversion 13
1.8 Expressions and Operators 15
1.9 Input Editing 22
1.10 Error Messages 23

2 Microsoft BASIC Commands and Statements 25

2.1 AUTO 29
2.2 CALL 30
2.3 CHAIN 31
2.4 CLEAR 34
2.5 CLOAD 35
2.6 CLOSE 36
2.7 COMMON 37
2.8 CONT 39
2.9 CSAVE 39
2.10 DATA 41
2.11 DEF FN 42
2.12 DEFINT/SNG/DBL/STR 43
2.13 DEF USR 44
2.14 DELETE 45
2.15 DIM 45
2.16 EDIT 46
2.17 END 50
2.18 ERASE 51

2.19 ERR and ERL Variables 51
2.20 ERROR 52
2.21 FIELD 54
2.22 FOR...NEXT 56
2.23 GET 59
2.24 GOSUB...RETURN 59
2.25 GOTO 60
2.26 IF...THEN[...ELSE] and IF...GOTO 61
2.27 INPUT 63
2.28 INPUT# 65
2.29 KILL 66
2.30 LET 66
2.31 LINE INPUT 67
2.32 LINE INPUT# 68
2.33 LIST 69
2.34 LLIST 70
2.35 LOAD 71
2.36 LPRINT and LPRINT USING 71
2.37 LSET and RSET 72
2.38 MERGE 73
2.39 MID$ 73
2.40 NAME 74
2.41 NEW 75
2.42 NULL 75
2.43 ON ERROR GOTO 76
2.44 ON...GOSUB and ON...GOTO 76
2.45 OPEN 77
2.46 OPTION BASE 78
2.47 OUT 79
2.48 POKE 79
2.49 PRINT 80
2.50 PRINT USING 82
2.51 PRINT# and PRINT# USING 86
2.52 PUT 89
2.53 RANDOMIZE 90
2.54 READ 91
2.55 REM 92
2.56 RENUM 93
2.57 RESTORE 94
2.58 RESUME 95
2.59 RUN 96
2.60 SAVE 97

2.61 STOP 98
2.62 SWAP 99
2.63 TRON/TROFF 99
2.64 WAIT 100
2.65 WHILE...WEND 101
2.66 WIDTH 102
2.67 WRITE 103
2.68 WRITE# 104

3 Microsoft BASIC Functions 105

3.1 ABS 107
3.2 ASC 108
3.3 ATN 108
3.4 CDBL 109
3.5 CHR$ 109
3.6 CINT 110
3.7 COS 110
3.8 CSNG 111
3.9 CVI, CVS, CVD 111
3.10 EOF 112
3.11 EXP 112
3.12 FIX 113
3.13 FRE 113
3.14 HEX$ 114
3.15 INKEY$ 114
3.16 INP 115
3.17 INPUT$ 115
3.18 INSTR 116
3.19 INT 116
3.20 LEFT$ 117
3.21 LEN 117
3.22 LOC 117
3.23 LOG 118
3.24 LPOS 118
3.25 MID$ 119
3.26 MKI$, MKS$, MKD$ 119
3.27 OCT$ 120
3.28 PEEK 120
3.29 POS 121
3.30 RIGHT$ 121

3.31 RND 122
3.32 SGN 122
3.33 SIN 123
3.34 SPACE$ 123
3.35 SPC 124
3.36 SQR 124
3.37 STR$ 125
3.38 STRING$ 125
3.39 TAB 125
3.40 TAN 126
3.41 USR 126
3.42 VAL 127
3.43 VARPTR 128

Appendices 129

A Error Codes and Error Messages 131

B Mathematical Functions 137

C ASCII Character Codes 139

D Microsoft BASIC Reserved Words 141

Index 143

Introduction

Microsoft® BASIC is the most extensive implementation of BASIC available for microprocessors. Microsoft BASIC meets the ANSI qualifications for BASIC, as set forth in document BSRX3.60-1978. Each release of Microsoft BASIC is compatible with previous versions.

How to Use This Manual

This manual is a reference for all implementations of Microsoft BASIC and for the Microsoft BASIC Compilers.

The manual is divided into three chapters plus four appendices. Chapter 1 covers a variety of topics, largely pertaining to data representation in Microsoft BASIC. Chapter 2 describes the syntax and semantics of every command and statement in Microsoft BASIC, ordered alphabetically. Chapter 3 describes all Microsoft BASIC intrinsic functions, also ordered alphabetically. The appendices contain a list of error messages and codes, a list of mathematical functions, a list of ASCII character codes, and a list of Microsoft BASIC reserved words.

Additional information about programming Microsoft BASIC is covered in the *Microsoft BASIC User's Guide.* The *User's Guide* describes the features of Microsoft BASIC which are implemented for your machine. It also contains information relevant to your operating system and helpful hints about such matters as data I/O and assembly language subroutines.

Syntax Notation

Wherever the syntax for a statement or command is given, the following rules apply:

CAPS	Items in capital letters must be input as shown.
< >	Items in lowercase letters enclosed in angle brackets (< >) are to be supplied by the user.
[ ]	Items in square brackets ([ ]) are optional.
. . .	Items followed by ellipses (. . .) may be repeated any number of times (up to the length of the line).
{ }	Braces indicate that the user has a choice between two or more entries. At least one of the entries enclosed in braces must be chosen unless the entries are also enclosed in square brackets.
|	Vertical bars separate the choices within braces. At least one of the entries separated by bars must be chosen unless the entries are also enclosed in square brackets.

All punctuation except angle brackets and square brackets (i.e., commas, parentheses, semicolons, hyphens, equal signs) must be included where shown.

Chapter 1

General Information about Microsoft BASIC

1.1 Initialization 5

1.2 Modes of Operation 5

1.3 Line Format 5

1.3.1 Line Numbers 6

1.4 Character Set 6

1.4.1 Control Characters 8

1.5 Constants 8

1.5.1 Single and Double Precision Form for Numeric Constants 10

1.6 Variables 11

1.6.1 Variable Names and Declaration Characters 11

1.6.2 Array Variables 12

1.6.3 Space Requirements 13

1.7 Type Conversion 13

1.8 Expressions and Operators 15

1.8.1 Arithmetic Operators 16

1.8.1.1 Integer Division and Modulus Arithmetic 16

1.8.1.2 Overflow and Division by Zero 17

1.8.2 Relational Operators 18

1.8.3 Logical Operators 19

1.8.4 Functional Operators 21
1.8.5 String Operators 21
1.9 Input Editing 22
1.10 Error Messages 23

1.1 Initialization

The procedure for initialization will vary with different implementations of Microsoft BASIC. Check the *Microsoft BASIC User's Guide* for your machine to determine how Microsoft BASIC is initialized with your operating system.

1.2 Modes of Operation

When Microsoft BASIC is initialized, it displays the prompt "Ok". "Ok" indicates Microsoft BASIC is at command level; that is, it is ready to accept commands. At this point, Microsoft BASIC may be used in either of two modes: direct mode or indirect mode.

In direct mode, Microsoft BASIC statements and commands are not preceded by line numbers. They are executed as they are entered. Results of arithmetic and logical operations may be displayed immediately and stored for later use, but the instructions themselves are lost after execution. Direct mode is useful for debugging and for using Microsoft BASIC as a "calculator" for quick computations that do not require a complete program.

Indirect mode is used for entering programs. Program lines are preceded by line numbers and are stored in memory. The program stored in memory is executed by entering the RUN command.

1.3 Line Format

Microsoft BASIC program lines have the following format (square brackets indicate optional input):

```
nnnnn BASIC statement[:BASIC statement...] <carriage return>
```

More than one BASIC statement may be placed on a line, but each must be separated from the last by a colon.

A Microsoft BASIC program line always begins with a line number and ends with a carriage return. A line may contain a maximum of 255 characters.

It is possible to extend a logical line over more than one physical line by using the <line feed> key. <line feed> lets you continue typing a logical line on the next physical line without entering a <carriage return>.

1.3.1 Line Numbers

Every Microsoft BASIC program line begins with a line number. Line numbers indicate the order in which the program lines are stored in memory. Line numbers are also used as references in branching and editing. Line numbers must be in the range 0 to 65529.

A period (.) may be used in EDIT, LIST, AUTO, and DELETE commands to refer to the current line.

1.4 Character Set

The Microsoft BASIC character set is comprised of alphabetic characters, numeric characters, and special characters.

The alphabetic characters in Microsoft BASIC are the uppercase and lowercase letters of the alphabet.

The Microsoft BASIC numeric characters include the digits 0 through 9.

In addition, the following special characters and terminal keys are recognized by Microsoft BASIC:

Character	Action
	Blank
=	Equals sign or assignment symbol
+	Plus sign
–	Minus sign
*	Asterisk or multiplication symbol
/	Slash or division symbol
^	Up arrow or exponentiation symbol
(	Left parenthesis
)	Right parenthesis
%	Percent
#	Number (or pound) sign
$	Dollar sign
!	Exclamation point
[	Left bracket
]	Right bracket
,	Comma
.	Period or decimal point
;	Semicolon
:	Colon
&	Ampersand
'	Single quotation mark (apostrophe)
?	Question mark
<	Less than
>	Greater than
\	Backslash or integer division symbol
@	At sign
_	Underscore
<rubout>	Deletes last character typed.
<escape>	Escapes edit mode subcommands. See Section 2.16.
<tab>	Moves print position to next tab stop. Tab stops are set every eight columns.
<line feed>	Moves to next physical line.
<carriage return>	Terminates input of a line.

1.4.1 Control Characters

Microsoft BASIC supports the following control characters:

Control Character	Action
Control-A	Enters edit mode on the line being typed.
Control-C	Interrupts program execution and returns to BASIC command level.
Control-G	Rings the bell at the terminal.
Control-H	Backspaces. Deletes the last character typed.
Control-I	Tabs to the next tab stop. Tab stops are set every eight columns.
Control-O	Halts program output while execution continues. A second Control-O resumes output.
Control-R	Lists the line that is currently being typed.
Control-S	Suspends program execution.
Control-Q	Resumes program execution after a Control-S.
Control-U	Deletes the line that is currently being typed.

1.5 Constants

Constants are the values Microsoft BASIC uses during execution. There are two types of constants: string and numeric.

A string constant is a sequence of up to 255 alphanumeric characters enclosed in double quotation marks.

Examples

```
"HELLO"
"$25,000.00"
"Number of Employees"
```

Numeric constants are positive or negative numbers. Microsoft BASIC numeric constants cannot contain commas. There are five types of numeric constants:

1. **Integer constants** — Whole numbers between -32768 and 32767. Integer constants do not contain decimal points.

2. **Fixed-point constants** — Positive or negative real numbers, i.e., numbers that contain decimal points.

3. **Floating-point constants** — Positive or negative numbers represented in exponential form (similar to scientific notation). A floating-point constant consists of an optionally signed integer or fixed-point number (the mantissa) followed by the letter E and an optionally signed integer (the exponent). The allowable range for floating-point constants is 10^{-38} to 10^{+38}.

   Examples

   ```
   235.988E-7 = .0000235988
   2359E6 = 2359000000
   ```

   (Double precision floating-point constants are denoted by the letter D instead of E. See Section 1.5.1.)

4. **Hex constants** — Hexadecimal numbers, denoted by the prefix &H.

   Examples

   ```
   &H76
   &H32F
   ```

5. Octal constants — Octal numbers, denoted by the prefix &O or &.

Examples

```
&O347
&1234
```

Note

The 8K version of Microsoft BASIC does not support hexadecimal or octal constants.

1.5.1 Single and Double Precision Form for Numeric Constants

Numeric constants may be either single precision or double precision numbers. Single precision numeric constants are stored with 7 digits of precision, and printed with up to 6 digits of precision. Double precision numeric constants are stored with 16 digits of precision and printed with up to 16 digits.

A single precision constant is any numeric constant that has one of the following characteristics:

1. Seven or fewer digits.
2. Exponential form using E.
3. A trailing exclamation point (!).

Examples

```
46.8
-1.09E-06
3489.0
22.5!
```

A double precision constant is any numeric constant that has one of these characteristics:

1. Eight or more digits.
2. Exponential form using D.
3. A trailing number sign (#).

Examples

```
345692811
-1.09432D-06
3489.0#
7654321.1234
```

1.6 Variables

Variables are names used to represent values used in a BASIC program. The value of a variable may be assigned explicitly by the programmer, or it may be assigned as the result of calculations in the program. Before a variable is assigned a value, its value is assumed to be zero.

1.6.1 Variable Names and Declaration Characters

Microsoft BASIC variable names may be any length. Up to 40 characters are significant. Variable names can contain letters, numbers, and the decimal point. However, the first character must be a letter. Special type declaration characters are also allowed—see below.

A variable name may not be a reserved word, but embedded reserved words are allowed. Reserved words include all Microsoft BASIC commands, statements, function names, and operator names. If a variable begins with FN, it is assumed to be a call to a user-defined function.

Variables may represent either a numeric value or a string. String variable names are written with a dollar sign ($) as the last char-

acter. For example: A$ = "SALES REPORT." The dollar sign is a variable type declaration character; that is, it "declares" that the variable will represent a string.

Numeric variable names may declare integer, single precision, or double precision values. The type declaration characters for these variable names are as follows:

%	Integer variable
!	Single precision variable
#	Double precision variable

The default type for a numeric variable name is single precision.

Examples of Microsoft BASIC variable names:

PI#	Declares a double precision value.
MINIMUM!	Declares a single precision value.
LIMIT%	Declares an integer value.
N$	Declares a string value.
ABC	Represents a single precision value.

There is a second method by which variable types may be declared. The Microsoft BASIC statements DEFINT, DEFSTR, DEFSNG, and DEFDBL may be included in a program to declare the types for certain variable names. These statements are described in detail in Section 2.12.

1.6.2 Array Variables

An array is a group or table of values referenced by the same variable name. Each element in an array is referenced by an array variable that is subscripted with an integer or an integer expression. An array variable name has as many subscripts as there are dimensions in the array. For example, V(10) would reference a value in a one-dimension array, T(1,4) would reference a value in a two-dimension array, and so on. The maximum number of dimensions for an array is 255. The maximum number of elements per dimension is 32,767.

1.6.3 Space Requirements

The following table lists only the number of bytes occupied by the values represented by the variable names. Additional requirements may vary according to implementation.

Variables	Type	Bytes
	Integer	2
	Single Precision	4
	Double Precision	8
Arrays	**Type**	**Bytes**
	Integer	2 per element
	Single Precision	4 per element
	Double Precision	8 per element
Strings		
3 bytes overhead plus the present contents of the string.		

1.7 Type Conversion

When necessary, Microsoft BASIC will convert a numeric constant from one type to another. The following rules and examples should be kept in mind.

1. If a numeric constant of one type is set equal to a numeric variable of a different type, the number will be stored as the type declared in the variable name. (If a string variable is set equal to a numeric value or vice versa, a "Type mismatch" error occurs.)

   Example

```
10 A% = 23.42
20 PRINT A%
RUN
 23
```

2. During expression evaluation, all of the operands in an arithmetic or relational operation are converted to the same degree of precision, i.e., that of the most precise operand. Also, the result of an arithmetic operation is returned to this degree of precision.

   Examples

```
10 D# = 6#/7
20 PRINT D#
RUN
 .8571428571428571
```

The arithmetic was performed in double precision and the result was returned in D# as a double precision value.

```
10 D = 6#/7
20 PRINT D
RUN
 .857143
```

The arithmetic was performed in double precision and the result was returned to D (single precision variable), rounded, and printed as a single precision value.

3. Logical operators (see Section 1.8.3) convert their operands to integers and return an integer result. Operands must be in the range -32768 to 32767 or an "Overflow" error occurs.

4. When a floating-point value is converted to an integer, the fractional portion is rounded.

   Example

```
10 C% = 55.88
20 PRINT C%
RUN
 56
```

5. If a double precision variable is assigned a single precision value, only the first seven digits (rounded) of the converted number will be valid. This is because only seven digits of accuracy were supplied with the single precision value. The absolute value of the difference between the printed double precision number and the original single precision value will be less than 6.3E-8 times the original single precision value.

   Example

```
10 A=2.04
20 B#=A
30 PRINT A;B#
RUN
 2.04 2.039999961853027
```

1.8 Expressions and Operators

An expression may be a string or numeric constant, a variable, or a combination of constants and variables with operators which produces a single value.

Operators perform mathematical or logical operations on values. The Microsoft BASIC operators may be divided into four categories:

1. Arithmetic
2. Relational
3. Logical
4. Functional

Each category is described in the following sections.

1.8.1 Arithmetic Operators

The arithmetic operators, in order of precedence, are:

Operator	Operation	Expression
^	Exponentiation	X^Y
-	Negation	-X
*,/	Multiplication, Floating-point Division	X*Y X/Y
+,-	Addition, Subtraction	X+Y

To change the order in which the operations are performed, use parentheses. Operations within parentheses are performed first. Inside parentheses, the usual order of operations is maintained.

Here are some sample algebraic expressions and their Microsoft BASIC counterparts.

Algebraic Expression	BASIC Expression	
X+2Y	X+Y*2	
$X-\frac{Y}{Z}$	X-Y/Z	
$\frac{XY}{Z}$	X*Y/Z	
$\frac{X+Y}{Z}$	(X+Y)/Z	
$(X^2)^Y$	(X^2)^Y	
X^{Y^Z}	X^(Y^Z)	
X(-Y)	X*(-Y)	Consecutive operators are separated by parentheses.

1.8.1.1 Integer Division and Modulus Arithmetic

Two additional operators are available in Microsoft BASIC: integer division and modulus arithmetic.

Integer division is denoted by the backslash (\). The operands are rounded to integers (must be in the range -32768 to 32767) before the division is performed, and the quotient is truncated to an integer.

Example

```
10\4 = 2
25.68\6.99 = 3
```

Integer division follows multiplication and floating-point division in order of precedence.

Modulus arithmetic is denoted by the operator MOD. Modulus arithmetic yields the integer value that is the remainder of an integer division.

Example

```
10.4 MOD 4=2          (10/4=2 with a remainder 2)
25.68 MOD 6.99=5      (26/7=3 with a remainder 5)
```

Modulus arithmetic follows integer division in order of precedence.

1.8.1.2 Overflow and Division by Zero

If, during the evaluation of an expression, division by zero is encountered, the "Division by zero" error message is displayed, machine infinity with the sign of the numerator is supplied as the result of the division, and execution continues. If the evaluation of an exponentiation operator results in zero being raised to a negative power, the "Division by zero" error message is displayed, positive machine infinity is supplied as the result of the exponentiation, and execution continues.

If overflow occurs, the "Overflow" error message is displayed, machine infinity with the algebraically correct sign is supplied as the result, and execution continues.

1.8.2 Relational Operators

Relational operators are used to compare two values. The result of the comparison is either "true" (-1) or "false" (0). This result may then be used to make a decision regarding program flow. (See "IF" statements, Section 2.26.)

The relational operators are:

Operator	Relation Tested	Example
=	Equality	X=Y
< >	Inequality	X< >Y
<	Less than	X<Y
>	Greater than	X>Y
<=	Less than or equal to	X<=Y
>=	Greater than or equal to	X>=Y

(The equal sign is also used to assign a value to a variable. See "LET," Section 2.30.)

When arithmetic and relational operators are combined in one expression, the arithmetic is always performed first. For example, the expression

```
X+Y<(T-1)/Z
```

is true if the value of X plus Y is less than the value of T-1 divided by Z.

Examples

```
IF SIN(X)<0 GOTO 1000
IF I MOD J<>0 THEN K=K+1
```

1.8.3 Logical Operators

Logical operators perform tests on multiple relations, bit manipulation, or Boolean operations. The logical operator returns a bitwise result which is either "true" (not zero) or "false" (zero). In an expression, logical operations are performed after arithmetic and relational operations. The outcome of a logical operation is determined as shown in Table 1. The operators are listed in order of precedence.

Table 1. Microsoft BASIC Relational Operators Truth Table

NOT			
	X	NOT X	
	1	0	
	0	1	
AND			
	X	Y	X AND Y
	1	1	1
	1	0	0
	0	1	0
	0	0	0
OR			
	X	Y	X OR Y
	1	1	1
	1	0	1
	0	1	1
	0	0	0
XOR			
	X	Y	X XOR Y
	1	1	0
	1	0	1
	0	1	1
	0	0	0
EQV			
	X	Y	X EQV Y
	1	1	1
	1	0	0
	0	1	0
	0	0	1
IMP			
	X	Y	X IMP Y
	1	1	1
	1	0	0
	0	1	1
	0	0	1

Just as the relational operators can be used to make decisions regarding program flow, logical operators can connect two or more relations and return a true or false value to be used in a decision (see "IF" statements, Section 2.26).

Example

```
IF D<200 AND F<4 THEN 80
IF I>10 OR K<0 THEN 50
IF NOT P THEN 100
```

Logical operators work by converting their operands to 16-bit, signed, two's complement integers in the range -32768 to 32767. (If the operands are not in this range, an error results.) If both operands are supplied as 0 or -1, logical operators return 0 or -1. The given operation is performed on these integers in bitwise fashion, i.e., each bit of the result is determined by the corresponding bits in the two operands.

Thus, it is possible to use logical operators to test bytes for a particular bit pattern. For instance, the AND operator may be used to "mask" all but one of the bits of a status byte at a machine I/O port. The OR operator may be used to "merge" two bytes to create a particular binary value. The following examples will help demonstrate how the logical operators work.

63 AND 16=16	63=binary 111111 and 16=binary 10000, so 63 AND 16=16.
15 AND 14=14	15=binary 1111 and 14=binary 1110, so 15 AND 14=14 (binary 1110).
-1 AND 8=8	-1=binary 1111111111111111 and 8=binary 1000, so -1 AND 8=8.
4 OR 2=6	4=binary 100 and 2=binary 10, so 4 OR 2=6 (binary 110).
10 OR 10=10	10=binary 1010, so 1010 OR 1010=1010 (decimal 10).
-1 OR -2=-1	-1=binary 1111111111111111 and -2=binary 1111111111111110, so -1 OR -2=-1. The bit complement of sixteen zeros is sixteen ones, which is the two's complement representation of -1.
NOT X=-(X+1)	The two's complement of any integer is the bit complement plus one.

1.8.4 Functional Operators

A function is used in an expression to call a predetermined operation that is to be performed on an operand. Microsoft BASIC has "intrinsic" functions that reside in the system, such as SQR (square root) or SIN (sine). All Microsoft BASIC intrinsic functions are described in Chapter 3.

Microsoft BASIC also allows "user-defined" functions that are written by the programmer. See "DEF FN," Section 2.11.

1.8.5 String Operators

Strings may be concatenated by using +.

Example

```
10 A$ = "FILE" : B$ = "NAME"
20 PRINT A$ + B$
30 PRINT "NEW" + A$ + B$
RUN
FILENAME
NEW FILENAME
```

Strings may be compared using the same relational operators that are used with numbers:

```
=    <>    <    >    <=    >=
```

String comparisons are made by taking one character at a time from each string and comparing the ASCII codes. If all the ASCII codes are the same, the strings are equal. If the ASCII codes differ, the lower code number precedes the higher. If during string comparison the end of one string is reached, the shorter string is said to be smaller. Leading and trailing blanks are significant.

Examples

```
"AA"<"AB"
"FILENAME" = "FILENAME"
"X&">"X#"
"CL">"CL"
"kg">"KG"
"SMYTH"<"SMYTHE"
B$<"9/12/78"  where
B$ = "8/12/78"
```

Thus, string comparisons can be used to test string values or to alphabetize strings. All string constants used in comparison expressions must be enclosed in quotation marks.

1.9 Input Editing

If an incorrect character is entered as a line is being typed, it can be deleted with the <RUBOUT> key or with Control-H. Rubout surrounds the deleted character(s) with backslashes. Control-H has the effect of backspacing over a character and erasing it. Once a character(s) has been deleted, simply continue typing the line as desired.

To delete a line that is in the process of being typed, type Control-U. A carriage return is executed automatically after the line is deleted.

To correct program lines for a program that is currently in memory, simply retype the line using the same line number. Microsoft BASIC will automatically replace the old line with the new line.

More sophisticated editing capabilities are provided. See "EDIT," Section 2.16.

To delete the entire program currently residing in memory, enter the NEW command. (See Section 2.41.) NEW is usually used to clear memory prior to entering a new program.

1.10 Error Messages

If an error causes program execution to terminate, an error message is printed. For a complete list of Microsoft BASIC error codes and error messages, see Appendix A.

Chapter 2

Microsoft BASIC Commands and Statements

Introduction 29

2.1 AUTO 29

2.2 CALL 30

2.3 CHAIN 31

2.4 CLEAR 34

2.5 CLOAD 35

2.6 CLOSE 36

2.7 COMMON 37

2.8 CONT 39

2.9 CSAVE 39

2.10 DATA 41

2.11 DEF FN 42

2.12 DEFINT/SNG/DBL/STR 43

2.13 DEF USR 44

2.14 DELETE 45

2.15 DIM 45

2.16 EDIT 46

2.17 END 50

2.18 ERASE 51
2.19 ERR and ERL Variables 51
2.20 ERROR 52
2.21 FIELD 54
2.22 FOR...NEXT 56
2.23 GET 59
2.24 GOSUB...RETURN 59
2.25 GOTO 60
2.26 IF...THEN[...ELSE] and IF...GOTO 61
2.27 INPUT 63
2.28 INPUT# 65
2.29 KILL 66
2.30 LET 66
2.31 LINE INPUT 67
2.32 LINE INPUT# 68
2.33 LIST 69
2.34 LLIST 70
2.35 LOAD 71
2.36 LPRINT and LPRINT USING 71
2.37 LSET and RSET 72
2.38 MERGE 73
2.39 MID$ 73
2.40 NAME 74
2.41 NEW 75
2.42 NULL 75

2.43 ON ERROR GOTO 76
2.44 ON . . . GOSUB and ON . . . GOTO 76
2.45 OPEN 77
2.46 OPTION BASE 78
2.47 OUT 79
2.48 POKE 79
2.49 PRINT 80
2.50 PRINT USING 82
2.51 PRINT# and PRINT# USING 86
2.52 PUT 89
2.53 RANDOMIZE 90
2.54 READ 91
2.55 REM 92
2.56 RENUM 93
2.57 RESTORE 94
2.58 RESUME 95
2.59 RUN 96
2.60 SAVE 97
2.61 STOP 98
2.62 SWAP 99
2.63 TRON/TROFF 99
2.64 WAIT 100
2.65 WHILE . . . WEND 101
2.66 WIDTH 102
2.67 WRITE 103
2.68 WRITE# 104

Microsoft BASIC commands and statements are described in this chapter. Each description is formatted as follows:

Syntax Shows the correct syntax for the instruction. See the "Introduction" to this manual for syntax notation.

Purpose Tells what the instruction is used for.

Remarks Describes in detail how the instruction is used.

Example Shows sample programs or program segments that demonstrate the use of the instruction.

Note Describes special cases or provides additional pertinent information.

2.1 AUTO

Syntax AUTO [<line number>[,<increment>]]

Purpose To generate a line number automatically after every carriage return.

Remarks AUTO begins numbering at <line number> and increments each subsequent line number by <increment>. The default for both values is 10. If <line number> is followed by a comma but <increment> is not specified, the last increment specified in an AUTO command is assumed.

If AUTO generates a line number that is already being used, an asterisk is printed after the number to warn the user that any input will replace the existing line. However, typing a carriage return immediately after the asterisk will save the line and generate the next line number.

AUTO is terminated by typing Control-C. The line in which Control-C is typed is not saved. After Control-C is typed, Microsoft BASIC returns to command level.

Example

AUTO 100,50	Generates line numbers 100, 150, 200 ....
AUTO	Generates line numbers 10, 20, 30, 40 ....

2.2 CALL

Syntax CALL <variable name>[(<argument list>)]

Purpose To call an assembly language subroutine.

Remarks The CALL statement is one way to transfer program flow to an external subroutine. (See also the USR function, Section 3.41)

<variable name> contains an address that is the starting point in memory of the subroutine. <variable name> may not be an array variable name. <argument list> contains the arguments that are passed to the external subroutine. <argument list> may contain only variables.

The CALL statement generates the same calling sequence used by Microsoft FORTRAN, Microsoft COBOL, and Microsoft BASIC Compilers.

Example

```
110 MYROUT = &HD000
120 CALL MYROUT(I,J,K)
 .
 .
 .
```

Note In a Microsoft BASIC Compiler program, line 110 is not required because the address of MYROUT will be assigned by the linking loader at load time.

2.3 CHAIN

Syntax CHAIN [MERGE]<filename>[,[<line number exp>] [,ALL][,DELETE <range>]]

Purpose To call a program and pass variables to it from the current program.

Remarks <filename> is the name of the program that is called.

<line number exp> is a line number or an expression that evaluates to a line number in the called program. It is the starting point for execution of the called program. If it is omitted, execution begins at the first line.

The COMMON statement may be used to pass variables (see Section 2.7).

Example 1

```
10  REM THIS PROGRAM DEMONSTRATES CHAINING USING COMMON TO PASS VARIABLES.
20  REM SAVE THIS MODULE ON DISK AS "PROG1" USING THE A OPTION.
30  DIM A$(2),B$(2)
40  COMMON A$(),B$()
50  A$(1) = "VARIABLES IN COMMON MUST BE ASSIGNED"
60  A$(2) = "VALUES BEFORE CHAINING."
70  B$(1) = " ": B$(2) = " "
80  CHAIN "PROG2"
90  PRINT: PRINT B$(1): PRINT: PRINT B$(2): PRINT
100 END
```

Example 2

```
10  REM THE STATEMENT "DIM A$(2),B$(2)" MAY
ONLY BE EXECUTED ONCE.
20  REM HENCE, IT DOES NOT APPEAR IN THIS
MODULE.
30  REM SAVE THIS MODULE ON THE DISK AS
"PROG2" USING THE A OPTION.
40  COMMON A$(),B$()
50  PRINT: PRINT A$(1);A$(2)
60  B$(1) = "NOTE HOW THE OPTION OF SPECIFY-
ING A STARTING LINE NUMBER"
70  B$(2) = "WHEN CHAINING AVOIDS THE DIMEN-
SION STATEMENT IN 'PROG1'."
80  CHAIN "PROG1",90
90  END
```

<line number exp> is not affected by a RENUM command.

With the ALL option, every variable in the current program is passed to the called program. If the ALL option is omitted, the current program must contain a COMMON statement to list the variables that are passed. See Section 2.7.

The MERGE option allows a subroutine to be brought into the BASIC program as an overlay. That is, a MERGE operation is performed with the current program and the called program. The called program must be an ASCII file if it is to be MERGEd.

After an overlay is brought in, it is usually desirable to delete it so that a new overlay may be brought in. To do this, use the DELETE option.

Example 3

```
10  REM THIS PROGRAM DEMONSTRATES CHAIN-
ING USING THE MERGE AND ALL OPTIONS.
20  REM SAVE THIS MODULE ON THE DISK AS
"MAINPRG".
30  A$ = "MAINPRG"
40  CHAIN MERGE "OVRLAY1",1010,ALL
50  END
```

```
1000  REM SAVE THIS MODULE ON THE DISK AS
"OVRLAY1" USING THE A OPTION.
1010  PRINT A$; "HAS CHAINED TO OVRLAY1."
1020  A$ = "OVRLAY1"
1030  B$ = "OVRLAY2"
1040  CHAIN MERGE "OVRLAY2",1010,ALL,
DELETE 1000-1050
1050  END
```

```
1000  REM SAVE THIS MODULE ON THE DISK AS
"OVRLAY2" USING THE A OPTION.
1010  PRINT A$;  "HAS CHAINED TO ";B$:"."
1020  END
```

The line numbers in <range> are affected by the RENUM command.

Note

The CHAIN statement with MERGE option leaves the files open and preserves the current OPTION BASE setting.

If the MERGE option is omitted, CHAIN does not preserve variable types or user-defined functions for use by the chained program. That is, any DEFINT, DEFSNG, DEFDBL, DEFSTR, or DEF FN statements containing shared variables must be restated in the chained program.

The Microsoft BASIC Compiler does not support the ALL, MERGE, DELETE, and <line number exp> options to CHAIN. Thus, the statement syntax is CHAIN <filename>. If you wish to maintain compatibility with Microsoft BASIC Compiler, it is recommended that COMMON be used to pass variables and that overlays not be used. The CHAIN statement leaves the files open during CHAINing.

When using the MERGE option, user-defined functions should be placed before any CHAIN MERGE statements in the program. Otherwise, the user-defined functions will be undefined after the merge is complete.

2.4 CLEAR

Syntax CLEAR [,[<expression1>][,<expression2>]]

Purpose To set all numeric variables to zero, all string variables to null, and to close all open files; and, optionally, to set the end of memory and the amount of stack space.

Remarks <expression1> is a memory location which, if specified, sets the highest location available for use by Microsoft BASIC.

<expression2> sets aside stack space for Microsoft BASIC. The default is 512 bytes or one-eighth of the available memory, whichever is smaller.

Note Microsoft BASIC allocates string space dynamically. An "Out of string space" error occurs only if there is no free memory left for Microsoft BASIC to use.

Microsoft BASIC Compiler supports the CLEAR statement with the restriction that <expression1> and <expression2> must be integer expressions. If a value of 0 is given for either expression, the appropriate default is used. The default stack size is 512 bytes, and the default top of memory is the current top of memory. The CLEAR statement performs the following actions:

Closes all files.
Clears all COMMON and user variables.
Resets the stack and string space.
Releases all disk buffers.

Examples

```
CLEAR
CLEAR ,32768
CLEAR ,,2000
CLEAR ,32768,2000
```

2.5 CLOAD

Syntax

CLOAD <filename>
CLOAD? <filename>
CLOAD* <array name>

Purpose

To load a program or an array from cassette tape into memory.

Remarks

CLOAD executes a NEW command before it loads the program from cassette tape. <filename> is the string expression or the first character of the string expression that was specified when the program was CSAVEd.

CLOAD? verifies tapes by comparing the program currently in memory with the file on tape that has the same filename. If they are the same, Microsoft BASIC prints "Ok". If not, Microsoft BASIC prints "NO GOOD".

CLOAD* loads a numeric array that has been saved on tape. The data on tape is loaded into the array called <array name> specified when the array was CSAVE*ed.

CLOAD and CLOAD? are always entered at command level as direct mode commands. CLOAD* may be entered at command level or used as a program statement. Make sure the array has been DIMensioned before it is loaded. Microsoft BASIC always returns to command level after a CLOAD, CLOAD?, or CLOAD* is executed. Before a CLOAD is executed, make sure the cassette recorder is properly connected and in the play mode, and the tape is positioned correctly.

See also "CSAVE," Section 2.9.

Note CLOAD and CSAVE are not included in all implementations of Microsoft BASIC.

Example

```
CLOAD "MAX2"
```

Loads file "MAX2" into memory.

2.6 CLOSE

Syntax

```
CLOSE [[#]<file number>[,[#]<file number...>]]
```

Purpose To conclude I/O to a disk file.

Remarks <file number> is the number under which the file was OPENed. A CLOSE with no arguments closes all open files.

The association between a particular file and file number terminates upon execution of a CLOSE statement. The file may then be reOPENed using the same or a different file number; likewise, that file number may now be reused to OPEN any file.

A CLOSE for a sequential output file writes the final buffer of output.

The END statement and the NEW command always CLOSE all disk files automatically. (STOP does not close disk files.)

Example See "Disk File Handling," in the *Microsoft BASIC User's Guide.*

2.7 COMMON

Syntax COMMON <list of variables>

Purpose To pass variables to a CHAINed program.

Remarks The COMMON statement is used in conjunction with the CHAIN statement. COMMON statements may appear anywhere in a program, though it is recommended that they appear at the beginning. The same variable cannot appear in more than one COMMON statement. Array variables are specified by appending "( )" to the variable name. If all variables are to be passed, use CHAIN with the ALL option and omit the COMMON statement.

Example

```
100 COMMON A,B,C,D(),G$
110 CHAIN "PROG3",10
.
.
.
```

Note Microsoft BASIC Compiler supports a modified version of the COMMON statement. The COMMON statement must appear in a program before any executable statements. The current nonexecutable statements are:

```
COMMON
DEFDBL, DEFINT, DEFSNG, DEFSTR
DIM
OPTION BASE
REM
$INCLUDE
```

Array variables used in a COMMON statement must be declared in a preceding DIM statement.

The standard form of the COMMON statement is referred to as blank COMMON. Microsoft FORTRAN Compiler-style named COMMON areas are also supported; however, the variables are not preserved across CHAINs. The syntax for named COMMON is:

```
COMMON /<name>/ <list of variables>
```

where <name> is comprised of 1 to 6 alphanumeric characters starting with a letter. This is useful for communicating with Microsoft FORTRAN Compiler and assembly language routines without having to explicitly pass parameters in the CALL statement.

The blank COMMON size and order of variables must be the same in the CHAINing and CHAINed programs. With Microsoft BASIC Compiler, the best way to insure this is to place all blank COMMON declarations in a single include file and use the $INCLUDE statement in each program.

For example:

```
MENU.BAS
        10 $INCLUDE COMDEF
         .
         .
         .
        1000 CHAIN "PROG1"

PROG1.BAS
        10 $INCLUDE COMDEF
         .
         .
         .
        2000 CHAIN "MENU"

COMDEF.BAS
        100 DIM A(100),B$(200)
        110 COMMON I,J,K,A()
        120 COMMON A$,B$(),X,Y,Z
```

2.8 CONT

Syntax CONT

Purpose To continue program execution after a Control-C has been typed or a STOP or END statement has been executed.

Remarks Execution resumes at the point where the break occurred. If the break occurred after a prompt from an INPUT statement, execution continues with the reprinting of the prompt ("?" or prompt string).

CONT is usually used in conjunction with STOP for debugging. When execution is stopped, intermediate values may be examined and changed using direct mode statements. Execution may be resumed with CONT or a direct mode GOTO, which resumes execution at a specified line number. CONT may be used to continue execution after an error has occurred.

CONT is invalid if the program has been edited during the break.

Example See "STOP," Section 2.61.

2.9 CSAVE

Syntax CSAVE <string expression>
CSAVE* <array variable name>

Purpose To save the program or an array currently in memory on cassette tape.

Remarks

Each program or array saved on tape is identified by a filename. When the command CSAVE <string expression> is executed, Microsoft BASIC saves the program currently in memory on tape and uses the first character in <string expression> as the filename. <string expression> may be more than one character, but only the first character is used for the filename.

When the command CSAVE*<array variable name> is executed, Microsoft BASIC saves the specified array on tape. The array must be a numeric array. The elements of a multidimensional array are saved with the leftmost subscript changing fastest. For example, when the 2-dimensional array specified by DIM A(2,2) is saved (see "DIM," Section 2.15), the array elements are saved in this order:

0,0
1,0
2,0
0,1
1,1
2,1
0,2
1,2
2,2

CSAVE may be used as a program statement or as a direct mode command.

Before a CSAVE or CSAVE* is executed, make sure the cassette recorder is properly connected and in the record mode.

See also "CLOAD," Section 2.5.

Note

CSAVE and CLOAD are not included in all implementations of Microsoft BASIC.

Example

```
CSAVE "TIMER"
```

Saves the program currently in memory on cassette under filename "TIMER".

2.10 DATA

Syntax DATA <list of constants>

Purpose To store the numeric and string constants that are accessed by the program's READ statement(s). (See "READ," Section 2.54.)

Remarks DATA statements are nonexecutable and may be placed anywhere in the program. A DATA statement may contain as many constants as will fit on a line (separated by commas). Any number of DATA statements may be used in a program. READ statements access DATA statements in order (by line number). The data contained therein may be thought of as one continuous list of items, regardless of how many items are on a line or where the lines are placed in the program.

<list of constants> may contain numeric constants in any format, i.e., fixed-point, floating-point, or integer. (No numeric expressions are allowed in the list.) String constants in DATA statements must be surrounded by double quotation marks only if they contain commas, colons, or significant leading or trailing spaces. Otherwise, quotation marks are not needed.

The variable type (numeric or string) given in the READ statement must agree with the corresponding constant in the DATA statement.

DATA statements may be reread from the beginning by use of the RESTORE statement (Section 2.57).

Example See "READ," Section 2.54.

2.11 DEF FN

Syntax

DEF FN<name>[(<parameter list>)] =
<function definition>

Purpose

To define and name a function that is written by the user.

Remarks

<name> must be a legal variable name. This name, preceded by FN, becomes the name of the function.

<parameter list> is comprised of those variable names in the function definition that are to be replaced when the function is called. The items in the list are separated by commas.

<function definition> is an expression that performs the operation of the function. It is limited to one line. Variable names that appear in this expression serve only to define the function; they do not affect program variables that have the same name. A variable name used in a function definition may or may not appear in the parameter list. If it does, the value of the parameter is supplied when the function is called. Otherwise, the current value of the variable is used.

The variables in the parameter list represent, on a one-to-one basis, the argument variables or values that will be given in the function call.

This statement may define either numeric or string functions. If a type is specified in the function name, the value of the expression is forced to that type before it is returned to the calling statement. If a type is specified in the function name and the argument type does not match, a "Type mismatch" error occurs.

A DEF FN statement must be executed before the function it defines may be called. If a function is called before it has been defined, an "Undefined user function" error occurs. DEF FN is illegal in the direct mode.

Example

```
410 DEF FNAB(X,Y) = X^3/Y^2
420 T = FNAB(I,J)
```

Line 410 defines the function FNAB. The function is called in line 420.

2.12 DEFINT/SNG/DBL/STR

Syntax

DEF<type> <range(s) of letters>

where <type> is INT, SNG, DBL, or STR

Purpose

To declare variable types as integer, single precision, double precision, or string.

Remarks

Any variable names beginning with the letter(s) specified in <range of letters> will be considered the type of variable specified in the <type> portion of the statement. However, a type declaration character always takes precedence over a DEF-type statement. (See "Variable Names and Declaration Characters," Section 1.6.1.)

If no type declaration statements are encountered, Microsoft BASIC assumes all variables without declaration characters are single precision variables.

Examples

10 DEFDBL L-P	All variables beginning with the letters L, M, N, O, and P will be double precision variables.
10 DEFSTR A	All variables beginning with the letter A will be string variables.
10 DEFINT I-N, W-Z	All variables beginning with the letters I, J, K, L, M, N, W, X, Y, Z will be integer variables.

2.13 DEF USR

Syntax

DEF USR[<digit>] = <integer expression>

Purpose

To specify the starting address of an assembly language subroutine.

Remarks

<digit> may be any digit from 0 to 9. The digit corresponds to the number of the USR routine whose address is being specified. If <digit> is omitted, DEF USR0 is assumed. The value of <integer expression> is the starting address of the USR routine. See "Assembly Language Subroutines," in the *Microsoft BASIC User's Guide.*

Any number of DEF USR statements may appear in a program to redefine subroutine starting addresses, thus allowing access to as many subroutines as necessary.

Example

```
.
.
.
200 DEF USR0 = 24000
210 X = USR0(Y^2/2.89)
.
.
.
```

2.14 DELETE

Syntax DELETE [<line number>][-<line number>]

Purpose To delete program lines.

Remarks Microsoft BASIC always returns to command level after a DELETE is executed. If <line number> does not exist, an "Illegal function call" error occurs.

Examples

DELETE 40	Deletes line 40.
DELETE 40-100	Deletes lines 40 through 100, inclusive.
DELETE -40	Deletes all lines up to and including line 40.

2.15 DIM

Syntax DIM <list of subscripted variables>

Purpose To specify the maximum values for array variable subscripts and allocate storage accordingly.

Remarks If an array variable name is used without a DIM statement, the maximum value of the array's subscript(s) is assumed to be 10. If a subscript is used that is greater than the maximum specified, a "Subscript out of range" error occurs. The minimum value for a subscript is always 0, unless otherwise specified with the OPTION BASE statement (see Section 2.46).

The DIM statement sets all the elements of the specified arrays to an initial value of zero.

Example

```
10 DIM A(20)
20 FOR I = 0 TO 20
30 READ A(I)
40 NEXT I
 .
 .
 .
```

2.16 EDIT

Syntax EDIT <line number>

Purpose To enter edit mode at the specified line.

Remarks In edit mode, it is possible to edit portions of a line without retyping the entire line. Upon entering edit mode, BASIC types the line number of the line to be edited, then it types a space and waits for an edit mode subcommand.

Edit Mode Subcommands

Edit mode subcommands are used to move the cursor or to insert, delete, replace, or search for text within a line. The subcommands are not echoed. However, most of the edit mode subcommands may be preceded by an integer which causes the command to be executed that number of times. When an integer is not specified, it is assumed to be 1.

Edit mode subcommands may be categorized according to the following functions:

1. Moving the cursor
2. Inserting text
3. Deleting text
4. Finding text
5. Replacing text

6. Ending and restarting edit mode
7. Entering edit mode from a syntax error

Note

In the descriptions that follow, <ch> represents any character, <text> represents a string of characters of arbitrary length, [i] represents an optional integer (the default is 1), and $ represents the Escape (or Altmode) key.

---

1. Moving the Cursor

   Space bar — Use the space bar to move the cursor to the right. [i]Space bar moves the cursor i spaces to the right. Characters are printed as you space over them.

   Rubout — In edit mode, [i]Rubout moves the cursor i spaces to the left (backspaces). Characters are printed as you backspace over them.

2. Inserting Text

   I — I<text>$ inserts <text> at the current cursor position. The inserted characters are printed on the terminal. To terminate insertion, press Escape. If a <carriage return> is typed during an Insert command, the effect is the same as pressing Escape and then <carriage return>. During an Insert command, the Rubout, Delete, or Underscore key on the terminal may be used to delete characters to the left of the cursor. Rubout will print out the characters as you backspace over them. Delete and Underscore will print an Underscore for each character that you backspace over. If an attempt is made to insert a character that will make the line longer than 255 characters, a bell (Control-G) sounds and the character is not printed.

X — The X subcommand extends the line. X moves the cursor to the end of the line, enters insert mode, and allows insertion of text as if an Insert command had been given. When you are finished extending the line, press Escape or carriage return.

3. Deleting Text

D — [i]D deletes i characters to the right of the cursor. The deleted characters are echoed between backslashes, and the cursor is positioned to the right of the last character deleted. If there are fewer than i characters to the right of the cursor, iD deletes the remainder of the line.

H — H deletes all characters to the right of the cursor and then automatically enters insert mode. H is useful for extending a line or replacing statements at the end of a line.

4. Finding Text

S — The subcommand [i]S<ch> searches for the ith occurrence of <ch> and positions the cursor before it. The character at the current cursor position is not included in the search. If <ch> is not found, the cursor stops at the end of the line. All characters passed over during the search are printed.

K — The subcommand [i]K<ch> is similar to [i]S<ch>, except all the characters passed over in the search are deleted. The cursor is positioned before <ch>, and the deleted characters are enclosed in backslashes.

5. Replacing Text

C — The subcommand C<ch> changes the next character to <ch>. If you wish to change the next i characters, use the subcommand iC, followed by as many characters as are specified by i. After the ith new character is typed, change mode is exited and you will return to edit mode.

6. Ending and Restarting Edit Mode

<cr>	Typing a <carriage return> prints the remainder of the line, saves the changes you made, and exits edit mode.
E	The E subcommand has the same effect as <carriage return>, except the remainder of the line is not printed.
Q	The Q subcommand returns to Microsoft BASIC command level, *without* saving any of the changes that were made to the line in edit mode.
L	The L subcommand lists the remainder of the line (saving any changes made so far) and repositions the cursor at the beginning of the line, still in edit mode. L is usually used to list the line when you first enter edit mode.
A	The A subcommand lets you begin editing a line over again. It restores the original line and repositions the cursor at the beginning.
Control-A	To enter edit mode on the line you are currently typing, type Control-A. Microsoft BASIC responds with a <carriage return>, an exclamation point (!), and a space. The cursor will be positioned at the first character in the line. Proceed by typing an edit mode subcommand. Remember, if you have just entered a line and wish to go back and edit it, the command "EDIT." will enter edit mode at the current line. (The line number symbol "." always refers to the current line.) If an unrecognizable command or illegal character is input to Microsoft BASIC while in edit mode, BASIC sends a Control-G (bell) to the terminal and the command or character is ignored.

7. Entering Edit Mode from a Syntax Error

When a syntax error is encountered during execution of a program, Microsoft BASIC automatically enters edit mode at the line that caused the error. For example:

```
10 K = 2(4)
RUN
?Syntax error in 10
10
```

When you finish editing the line and press <carriage return> (or the E subcommand), Microsoft BASIC reinserts the line. This causes all variable values to be lost. To preserve the variable values for examination, first exit edit mode with the Q subcommand. Microsoft BASIC will return to command level, and all variable values will be preserved.

2.17 END

Syntax END

Purpose To terminate program execution, close all files, and return to command level.

Remarks END statements may be placed anywhere in the program to terminate execution. Unlike the STOP statement, END does not cause a "Break in line nnnnn" message to be printed. An END statement at the end of a program is optional. Microsoft BASIC always returns to command level after an END is executed.

Example

```
520 IF K>1000 THEN END ELSE GOTO 20
```

2.18 ERASE

Syntax ERASE <list of array variables>

Purpose To eliminate arrays from a program.

Remarks Arrays may be redimensioned after they are ERASEd, or the previously allocated array space in memory may be used for other purposes. If an attempt is made to redimension an array without first ERASEing it, a "Redimensioned array" error occurs.

Microsoft BASIC Compiler does not support ERASE.

Example

```
.
.
.
450 ERASE A,B
460 DIM B(99)
.
.
.
```

2.19 ERR and ERL Variables

When an error handling routine is entered, the variable ERR contains the error code for the error and the variable ERL contains the line number of the line in which the error was detected. The ERR and ERL variables are usually used in IF...THEN statements to direct program flow in the error handling routine.

If the statement that caused the error was a direct mode statement, ERL will contain 65535. To test whether an error occurred in a direct statement, use IF 65535=ERL THEN .... Otherwise, use

```
IF ERR = error code THEN ...
IF ERL = line number THEN ...
```

If the line number is not on the right side of the relational operator, it cannot be renumbered with RENUM. Because ERL and ERR are reserved variables, neither may appear to the left of the equal sign in a LET (assignment) statement. Microsoft BASIC error codes are listed in Appendix A.

2.20 ERROR

Syntax ERROR <integer expression>

Purpose To simulate the occurrence of a BASIC error, or to allow error codes to be defined by the user.

Remarks The value of <integer expression> must be greater than 0 and less than 255. If the value of <integer expression> equals an error code already in use by BASIC (see Appendix A), the ERROR statement will simulate the occurrence of that error and the corresponding error message will be printed. (See Example 1.)

To define your own error code, use a value that is greater than any used by MS-BASIC error codes. (It is preferable to use the highest available values, so compatibility may be maintained when more error codes are added to MS-BASIC.) This user-defined error code may then be conveniently handled in an error handling routine. (See Example 2.)

If an ERROR statement specifies a code for which no error message has been defined, MS-BASIC responds with the "Unprintable error" error message. Execution of an ERROR statement for which there is no error handling routine causes an error message to be printed and execution to halt.

Example 1

```
LIST
10 S = 10
20 T = 5
30 ERROR S + T
40 END
Ok
RUN
String too long in line 30
```

Or, in direct mode:

```
Ok
ERROR 15                    (You type this line.)
String too long             (BASIC types this line.)
Ok
```

Example 2

```
.
.
.
110 ON ERROR GOTO 400
120 INPUT "WHAT IS YOUR BET";B
130 IF B>5000 THEN ERROR 210
.
.
.
400 IF ERR = 210 THEN PRINT "HOUSE LIMIT IS
$5000"
410 IF ERL = 130 THEN RESUME 120
.
.
.
```

2.21 FIELD

Syntax

FIELD [#]<file number>,<field width> AS <string variable>...

Purpose

To allocate space for variables in a random file buffer.

Remarks

Before a GET statement or PUT statement can be executed, a FIELD statement must be executed to format the random file buffer.

<file number> is the number under which the file was OPENed. <field width> is the number of characters to be allocated to <string variable>. For example,

```
FIELD 1,20 AS N$,10 AS ID$,40 AS ADD$
```

allocates the first 20 positions (bytes) in the random file buffer to the string variable N$, the next 10 positions to ID$, and the next 40 positions to ADD$. FIELD does not place any data in the random file buffer. (See "LSET and RSET," Section 2.37, and "GET," Section 2.23.)

The total number of bytes allocated in a FIELD statement must not exceed the record length that was specified when the file was OPENed. Otherwise, a "Field overflow" error occurs. (The default record length is 128 bytes.)

Any number of FIELD statements may be executed for the same file. All FIELD statements that have been executed will remain in effect at the same time.

Note

Do not use a FIELDed variable name in an INPUT or LET statement. Once a variable name is FIELDed, it points to the correct place in the ran-

dom file buffer. If a subsequent INPUT or LET statement with that variable name is executed, the variable's pointer is moved to string space.

Example 1

```
FIELD 1,20 AS N$,10 AS ID$,40 AS ADD$
```

Allocates the first 20 positions (bytes) in the random file buffer to the string variable N$, the next 10 positions to ID$, and the next 40 positions to ADD$. FIELD does NOT place any data in the random file buffer. (See also "GET," Section 2.23, and "LSET and RSET," Section 2.37.)

Example 2

```
10 OPEN "R,"#1,"A:PHONELST",35
15 FIELD #1,2 AS RECNBR$,33 AS DUMMY$
20 FIELD #1,25 AS NAMES,10 AS PHONENBR$
25 GET #1
30 TOTAL = CVI(RECNBR)$
35 FOR I = 2 TO TOTAL
40 GET #1, I
45 PRINT NAMES, PHONENBR$
50 NEXT I
```

Illustrates a multiple defined FIELD statement. In statement 15, the 35-byte field is defined for the first record to keep track of the number of records in the file. In the next loop of statements (35–50), statement 20 defines the field for individual names and phone numbers.

Example 3

```
10 FOR LOOP% = 0 TO 7
20 FIELD #1,(LOOP% * 16) AS OFFSETS,16 AS
A$(LOOP%)
30 NEXT LOOP%
```

Shows the construction of a FIELD statement using an array of elements of equal size. The result is equivalent to the single declaration:

```
FIELD#1,16 AS A$(0),16 AS A$(1),. . .,16 AS A$(6),16
AS A$(7)
```

Example 4

```
10 DIM SIZE% (NUMB%): REM ARRAY OF FIELD
SIZES
20 FOR LOOP% = 0 TO NUMB%:READ SIZE%
(LOOP%):.NEXT LOOP%
30 DATA 9,10,12,21,41
 .
 .
 .
120 DIM A$(NUMB%): REM ARRAY OF FIELDED
VARIABLES
130 OFFSET% = 0
140 FOR LOOP% = 0 TO NUMB%
150 FIELD #1, OFFSET% AS OFFSET$, SIZE%
(LOOP%) AS A$(LOOP%)
160 OFFSET% = OFFSET% + SIZE%(LOOP%)
170 NEXT LOOP%
```

Creates a field in the same manner as Example 3. However, the element size varies with each element. The equivalent declaration is:

```
FIELD #1,SIZE%(0) AS A$(0),SIZE%(1) AS A$(1),. . .
SIZE%(NUMB%) AS A$(NUMB%)
```

2.22 FOR...NEXT

Syntax

```
FOR <variable> = x TO y [STEP z]
 .
 .
 .
NEXT [<variable>][,<variable>...]
```

where x, y, and z are numeric expressions.

Purpose

To allow a series of instructions to be performed in a loop a given number of times.

Remarks

<variable> is used as a counter. The first numeric expression (x) is the initial value of the counter. The second numeric expression (y) is the final value of the counter. The program lines following

the FOR statement are executed until the NEXT statement is encountered. Then the counter is adjusted by the amount specified by STEP. A check is performed to see if the value of the counter is now greater than the final value (y). If it is not greater, Microsoft BASIC branches back to the statement after the FOR statement and the process is repeated. If it is greater, execution continues with the statement following the NEXT statement. This is a FOR...NEXT loop.

If STEP is not specified, the increment is assumed to be one. If STEP is negative, the final value of the counter is set to be less than the initial value. The counter is decreased each time through the loop. The loop is executed until the counter is less than the final value.

The counter must be an integer or single precision numeric constant. If a double precision numeric constant is used, a "Type mismatch" error will result.

The body of the loop is skipped if the initial value of the loop times the sign of the STEP exceeds the final value times the sign of the STEP.

Nested Loops

FOR...NEXT loops may be nested; that is, a FOR...NEXT loop may be placed within the context of another FOR...NEXT loop. When loops are nested, each loop must have a unique variable name as its counter. The NEXT statement for the inside loop must appear before that for the outside loop. If nested loops have the same end point, a single NEXT statement may be used for all of them.

The variable(s) in the NEXT statement may be omitted, in which case the NEXT statement will match the most recent FOR statement. If a NEXT statement is encountered before its corresponding FOR statement, a "NEXT without FOR" error message is issued and execution is terminated.

Example 1

```
10 K=10
20 FOR I=1 TO K STEP 2
30 PRINT I;
40 K=K+10
50 PRINT K
60 NEXT
RUN
 1  20
 3  30
 5  40
 7  50
 9  60
Ok
```

Example 2

```
10 J=0
20 FOR I=1 TO J
30 PRINT I
40 NEXT I
```

In this example, the loop does not execute because the initial value of the loop exceeds the final value.

Example 3

```
10 I=5
20 FOR I=1 TO I+5
30 PRINT I;
40 NEXT
RUN
 1  2  3  4  5  6  7  8  9  10
Ok
```

In this example, the loop executes ten times. The final value for the loop variable is always set before the initial value is set.

Note

Previous versions of Microsoft BASIC set the initial value of the loop variable before setting the final value; i.e., the above loop would have executed six times.

2.23 GET

Syntax GET [#]<file number>[,<record number>]

Purpose To read a record from a random disk file into a random buffer.

Remarks <file number> is the number under which the file was OPENed. If <record number> is omitted, the next record (after the last GET) is read into the buffer. The largest possible record number is 32767.

Example See "Disk File Handling," in the *Microsoft BASIC User's Guide.*

Note After a GET statement has been executed, INPUT# and LINE INPUT# may be executed to read characters from the random file buffer.

2.24 GOSUB...RETURN

Syntax

```
GOSUB <line number>
.
.
.
RETURN
```

Purpose To branch to and return from a subroutine.

Remarks <line number> is the first line of the subroutine.

A subroutine may be called any number of times in a program. A subroutine also may be called from within another subroutine. Such nesting of subroutines is limited only by available memory.

The RETURN statement(s) in a subroutine cause Microsoft BASIC to branch back to the statement following the most recent GOSUB statement. A subroutine may contain more than one RETURN statement, should logic dictate a return at different points in the subroutine. Subroutines may appear anywhere in the program, but it is recommended that the subroutine be readily distinguishable from the main program. To prevent inadvertent entry into the subroutine, precede it with a STOP, END, or GOTO statement that directs program control around the subroutine.

Example

```
10 GOSUB 40
20 PRINT "BACK FROM SUBROUTINE"
30 END
40 PRINT "SUBROUTINE";
50 PRINT " IN";
60 PRINT " PROGRESS"
70 RETURN
RUN
SUBROUTINE IN PROGRESS
BACK FROM SUBROUTINE
Ok
```

2.25 GOTO

Syntax

GOTO <line number>

Purpose

To branch unconditionally out of the normal program sequence to a specified line number.

Remarks

If <line number> is an executable statement, that statement and those following are executed. If it is a nonexecutable statement, execution proceeds at the first executable statement encountered after <line number>.

Example

```
LIST
10 READ R
20 PRINT "R =";R,
30 A=3.14*R^2
40 PRINT "AREA =";A
50 GOTO 10
60 DATA 5,7,12
Ok
RUN
R =  5        AREA =  78.5
R =  7        AREA =  153.86
R =  12       AREA =  452.16
?Out of data in 10
Ok
```

2.26 IF...THEN[...ELSE] and IF...GOTO

Syntax 1

IF <expression> THEN {<statement(s)>|<line number>} [ELSE {<statement(s)>|<line number>}]

Syntax 2

IF <expression> GOTO <line number> [ELSE {<statement(s)>|<line number>}]

Purpose

To make a decision regarding program flow based on the result returned by an expression.

Remarks

If the result of <expression> is not zero, the THEN or GOTO clause is executed. THEN may be followed by either a line number for branching or one or more statements to be executed. GOTO is always followed by a line number. If the result of <expression> is zero, the THEN or GOTO clause is ignored and the ELSE clause, if present, is executed. Execution continues with the next executable statement. A comma is allowed before THEN.

Nesting of IF Statements

IF...THEN...ELSE statements may be nested. Nesting is limited only by the length of the line.

For example,

```
IF X>Y THEN PRINT "GREATER" ELSE IF Y>X
THEN PRINT "LESS THAN" ELSE PRINT "EQUAL"
```

is a legal statement. If the statement does not contain the same number of ELSE and THEN clauses, each ELSE is matched with the closest unmatched THEN. For example,

```
IF A = B THEN IF B = C THEN PRINT "A = C" ELSE
PRINT "A<>C"
```

will not print "A<>C" when A<>B.

If an IF...THEN statement is followed by a line number in direct mode, an "Undefined line" error results, unless a statement with the specified line number had previously been entered in indirect mode.

Note

When using IF to test equality for a value that is the result of a floating-point computation, remember that the internal representation of the value may not be exact. Therefore, the test should be against the range over which the accuracy of the value may vary. For example, to test a computed variable A against the value 1.0, use:

```
IF ABS (A-1.0)<1.0E-6 THEN ...
```

This test returns true if the value of A is 1.0 with a relative error of less than 1.0E-6.

Example 1

```
200 IF I THEN GET#1,I
```

This statement GETs record number I if I is not zero.

Example 2

```
100 IF(I<20)*(I>10) THEN DB = 1979-1:GOTO 300
110 PRINT "OUT OF RANGE"
 .
 .
 .
```

In this example, a test determines if I is greater than 10 and less than 20. If I is in this range, DB is calculated and execution branches to line 300. If I is not in this range, execution continues with line 110.

Example 3

```
210 IF IOFLAG THEN PRINT A$ ELSE LPRINT A$
```

This statement causes printed output to go either to the terminal or the line printer, depending on the value of the variable IOFLAG. If IOFLAG is zero, output goes to the line printer; otherwise, output goes to the terminal.

2.27 INPUT

Syntax

INPUT[;] [<"prompt string">;]<list of variables>

Purpose

To allow input from the terminal during program execution.

Remarks

When an INPUT statement is encountered, program execution pauses and a question mark is printed to indicate the program is waiting for data. If <"prompt string"> is included, the string is printed before the question mark. The required data is then entered at the terminal.

A comma may be used instead of a semicolon after the prompt string to suppress the question mark. For example, the statement INPUT "ENTER BIRTHDATE",B$ will print the prompt with no question mark.

If INPUT is immediately followed by a semicolon, then the carriage return typed to input data does not echo a carriage return/line feed sequence.

The data that is entered is assigned to the variable(s) given in <variable list>. The number of data items supplied must be the same as the number of variables in the list. Data items are separated by commas.

The variable names in the list may be numeric or string variable names (including subscripted variables). The type of each data item that is input must agree with the type specified by the variable name. (Strings input to an INPUT statement need not be surrounded by quotation marks.)

Responding to INPUT with too many or too few items or with the wrong type of value (numeric instead of string, etc.) causes the message "?Redo from start" to be printed. No assignment of input values is made until an acceptable response is given.

Examples

```
10 INPUT X
20 PRINT X "SQUARED IS" X^2
30 END
RUN
? 5                          (The 5 was typed in by the
                              user in response to the ques-
                              tion mark.)
  5 SQUARED IS 25
Ok
```

```
LIST
10 PI = 3.14
20 INPUT "WHAT IS THE RADIUS";R
30 A = PI*R^2
40 PRINT "THE AREA OF THE CIRCLE? IS";A
50 PRINT
60 GOTO 20
Ok
RUN
WHAT IS THE RADIUS? 7.4      (User types 7.4)
THE AREA OF THE CIRCLE IS 171.946

WHAT IS THE RADIUS?
etc.
```

2.28 INPUT#

Syntax INPUT#<file number>,<variable list>

Purpose To read data items from a sequential disk file and assign them to program variables.

Remarks <file number> is the number used when the file was OPENed for input. <variable list> contains the variable names that will be assigned to the items in the file. (The variable type must match the type specified by the variable name.) With INPUT#, no question mark is printed, as with INPUT.

The data items in the file should appear just as they would if data were being typed in response to an INPUT statement. With numeric values, leading spaces, carriage returns, and line feeds are ignored. The first character encountered that is not a space, carriage return, or line feed is assumed to be the start of a number. The number terminates on a space, carriage return, line feed, or comma.

If Microsoft BASIC is scanning the sequential data file for a string item, leading spaces, carriage returns, and line feeds are also ignored. The first character encountered that is not a space, carriage return, or line feed is assumed to be the start of a string item. If this first character is a quotation mark ("), the string item will consist of all characters read between the first quotation mark and the second. Thus, a quoted string may not contain a quotation mark as a character. If the first character of the string is not a quotation mark, the string is an unquoted string, and will terminate on a comma, carriage return, or line feed (or after 255 characters have been read). If end-of-file is reached when a numeric or string item is being INPUT, the item is terminated.

Example See "Disk File Handling," in the *Microsoft BASIC User's Guide.*

2.29 KILL

Syntax KILL <filename>

Purpose To delete a file from disk.

Remarks If a KILL statement is given for a file that is currently OPEN, a "File already open" error occurs.

KILL is used for all types of disk files: program files, random data files, and sequential data files.

Example

```
200 KILL "DATA1.DAT"
```

See also "Disk File Handling," in the *Microsoft BASIC User's Guide.*

2.30 LET

Syntax [LET]<variable> = <expression>

Purpose To assign the value of an expression to a variable.

Remarks Notice the word LET is optional; i.e., the equal sign is sufficient for assigning an expression to a variable name.

Example

```
110 LET D = 12
120 LET E = 12^2
130 LET F = 12^4
140 LET SUM = D + E + F
 .
 .
 .
```

or

```
110 D = 12
120 E = 12^2
130 F = 12^4
140 SUM = D + E + F
 .
 .
 .
```

2.31 LINE INPUT

Syntax

LINE INPUT[;] [<"prompt string">;]
<string variable>

Purpose

To input an entire line (up to 254 characters) to a string variable, without the use of delimiters.

Remarks

<"prompt string"> is a string literal that is printed at the terminal before input is accepted. A question mark is not printed unless it is part of <"prompt string">. All input from the end of <"prompt string"> to the carriage return is assigned to <string variable>. However, if a line feed/carriage return sequence (this order only) is encountered, both characters are echoed; but the carriage return is ignored, the line feed is put into <string variable>, and data input continues.

If LINE INPUT is immediately followed by a semicolon, then the carriage return typed by the user to end the input line does not echo a carriage return/line feed sequence at the terminal.

A LINE INPUT statement may be aborted by typing Control-C. Microsoft BASIC will return to command level and type "Ok". Typing CONT resumes execution at the LINE INPUT.

Example See "LINE INPUT#," Section 2.32.

2.32 LINE INPUT#

Syntax LINE INPUT#<file number>,<string variable>

Purpose To read an entire line (up to 254 characters), without delimiters, from a sequential disk data file to a string variable.

Remarks <file number> is the number under which the file was OPENed. <string variable> is the variable name to which the line will be assigned. LINE INPUT# reads all characters in the sequential file up to a carriage return. It then skips over the carriage return/line feed sequence. The next LINE INPUT# reads all characters up to the next carriage return. (If a line feed/carriage return sequence is encountered, it is preserved.)

LINE INPUT# is especially useful if each line of a data file has been broken into fields, or if a Microsoft BASIC program saved in ASCII format is being read as data by another program. (See "SAVE," Section 2.60.)

Example

```
10 OPEN "0",1,"LIST"
20 LINE INPUT "CUSTOMER INFORMATION?";C$
30 PRINT #1, C$
40 CLOSE 1
50 OPEN "I",1,"LIST"
60 LINE INPUT #1, C$
70 PRINT C$
80 CLOSE 1
RUN
CUSTOMER INFORMATION? LINDA JONES  234,4
MEMPHIS
LINDA JONES  234,4  MEMPHIS
Ok
```

2.33 LIST

Syntax 1 LIST[<line number>]

Syntax 2 LIST [<line number>][-[<line number>]]

Purpose To list all or part of the program currently in memory at the terminal.

Remarks Microsoft BASIC always returns to command level after a LIST is executed.

Syntax 1

If <line number> is omitted, the program is listed beginning at the lowest line number. (Listing is terminated either when the end of the program is reached or by typing Control-C.) If <line number> is included, only the specified line will be listed.

Syntax 2

This syntax allows the following options:

1. If only the first <line number> is specified, that line and all higher-numbered lines are listed.

2. If only the second <line number> (i.e., [-[<line number>]]) is specified, all lines from the beginning of the program through that line are listed.
3. If both <line number(s)> are specified, the entire range is listed.

Examples

Syntax 1

LIST	Lists the program currently in memory.
LIST 500	Lists line 500.

Syntax 2

LIST 150-	Lists all lines from 150 to the end.
LIST -1000	Lists all lines from the lowest number through 1000.
LIST 150-1000	Lists lines 150 through 1000, inclusive.

2.34 LLIST

Syntax LLIST [<line number>[-[<line number>]]]

Purpose To list all or part of the program currently in memory at the line printer.

Remarks LLIST assumes a 132-character-wide printer.

Microsoft BASIC always returns to command level after an LLIST is executed. The options for LLIST are the same as for LIST, Syntax 2.

Note LLIST and LPRINT are not included in all implementations of Microsoft BASIC.

Example See the examples for "LIST," Syntax 2.

2.35 LOAD

Syntax LOAD <filename>[,R]

Purpose To load a file from disk into memory.

Remarks <filename> is the name that was used when the file was SAVEd. (Your operating system may append a default filename extension if one was not supplied in the SAVE command. Refer to "Microsoft BASIC Disk I/O," in the *Microsoft BASIC User's Guide,* for information about possible filename extensions under your operating system.)

The R option automatically runs the program after it has been loaded.

LOAD closes all open files and deletes all variables and program lines currently residing in memory before it loads the designated program. However, if the R option is used with LOAD, the program is RUN after it is LOADed, and all open data files are kept open. Thus, LOAD with the R option may be used to chain several programs (or segments of the same program). Information may be passed between the programs using their disk data files.

Example
```
LOAD "STRTRK",R
LOAD "B:MYPROG"
```

2.36 LPRINT and LPRINT USING

Syntax
```
LPRINT [<list of expressions>]
LPRINT USING <string exp>;<list of expressions>
```

Purpose To print data at the line printer.

Remarks Same as PRINT and PRINT USING, except output goes to the line printer. See Section 2.49 and Section 2.50.

LPRINT assumes a 132-character-wide printer.

Note LPRINT and LLIST are not included in all implementations of Microsoft BASIC.

2.37 LSET and RSET

Syntax

LSET <string variable> = <string expression>
RSET <string variable> = <string expression>

Purpose To move data from memory to a random file buffer (in preparation for a PUT statement).

Remarks If <string expression> requires fewer bytes than were FIELDed to <string variable>, LSET left-justifies the string in the field, and RSET right-justifies the string. (Spaces are used to pad the extra positions.) If the string is too long for the field, characters are dropped from the right. Numeric values must be converted to strings before they are LSET or RSET. See "MKI$, MKS$, MKD$," Section 3.26.

Examples

```
150 LSET A$ = MKS$(AMT)
160 LSET D$ = DESC($)
```

See also "Disk File Handling," in the *Microsoft BASIC User's Guide.*

Note LSET or RSET may also be used with a nonfielded string variable to left-justify or right-justify a string in a given field. For example, the program lines

```
110 A$ = SPACE$(20)
120 RSET A$ = N$
```

right-justify the string N$ in a 20-character field. This can be very handy for formatting printed output.

2.38 MERGE

Syntax MERGE <filename>

Purpose To merge a specified disk file into the program currently in memory.

Remarks <filename> is the name used when the file was SAVEd. (Your operating system may append a default filename extension if one was not supplied in the SAVE command. Refer to "Disk File Handling," in the *Microsoft BASIC User's Guide,* for information about possible filename extensions under your operating system.) The file must have been SAVEd in ASCII format. (If not, a "Bad file mode" error occurs.)

If any lines in the disk file have the same line numbers as lines in the program in memory, the lines from the file on disk will replace the corresponding lines in memory. (MERGEing may be thought of as "inserting" the program lines on disk into the program in memory.)

Microsoft BASIC always returns to command level after executing a MERGE command.

Example MERGE "NUMBRS"

2.39 MID$

Syntax MID$(<string exp1>,n[,m]) = <string exp2>

where n and m are integer expressions and <string exp1> and <string exp2> are string expressions.

Purpose To replace a portion of one string with another string.

Remarks The characters in <string exp1>, beginning at position n, are replaced by the characters in <string exp2>. The optional "m" refers to the number of characters from <string exp2> that will be used in the replacement. If "m" is omitted, all of <string exp2> is used. However, regardless of whether "m" is omitted or included, the replacement of characters never goes beyond the original length of <string exp1>.

Example

```
10 A$ = "KANSAS CITY, MO"
20 MID$(A$,14) = "KS"
30 PRINT A$
RUN
KANSAS CITY, KS
```

MID$ is also a function that returns a substring of a given string. See Section 3.25.

2.40 NAME

Syntax NAME <old filename> AS <new filename>

Purpose To change the name of a disk file.

Remarks <old filename> must exist and <new filename> must not exist; otherwise, an error will result. After a NAME command, the file exists on the same disk, in the same area of disk space, with the new name.

Example

```
Ok
NAME "ACCTS" AS "LEDGER"
Ok
```

In this example, the file that was formerly named ACCTS will now be named LEDGER.

2.41 NEW

Syntax NEW

Purpose To delete the program currently in memory and clear all variables.

Remarks NEW is entered at command level to clear memory before entering a new program. Microsoft BASIC always returns to command level after a NEW is executed.

Example NEW

2.42 NULL

Syntax NULL <integer expression>

Purpose To set the number of nulls to be printed at the end of each line.

Remarks For 10 character-per-second tape punches, <integer expression> should be >=3. When tapes are not being punched, <integer expression> should be 0 or 1 for Teletype® and Teletype-compatible terminal screens. <integer expression> should be 2 or 3 for 30 CPS hard copy printers. The default value is 0.

Example

```
Ok
NULL 2
Ok
100 INPUT X
200 IF X<50 GOTO 800
 .
 .
 .
```

Two null characters will be printed after each line.

2.43 ON ERROR GOTO

Syntax ON ERROR GOTO <line number>

Purpose To enable error handling and specify the first line of the error handling routine.

Remarks Once error handling has been enabled, all errors detected, including direct mode errors (e.g., syntax errors), will cause a jump to the specified error handling routine. If <line number> does not exist, an "Undefined line" error results.

To disable error handling, execute an ON ERROR GOTO 0. Subsequent errors will print an error message and halt execution. An ON ERROR GOTO 0 statement that appears in an error handling routine causes Microsoft BASIC to stop and print the error message for the error that caused the trap. It is recommended that all error handling routines execute an ON ERROR GOTO 0 if an error is encountered for which there is no recovery action.

Note If an error occurs during execution of an error handling routine, that error message is printed and execution terminates. Error trapping does not occur within the error handling routine.

Example

```
10 ON ERROR GOTO 1000
```

2.44 ON...GOSUB and ON...GOTO

Syntax ON <expression> GOTO <list of line numbers>
ON <expression> GOSUB <list of line numbers>

Purpose To branch to one of several specified line numbers, depending on the value returned when an expression is evaluated.

Remarks The value of <expression> determines which line number in the list will be used for branching. For example, if the value is three, the third line number in the list will be the destination of the branch. (If the value is a noninteger, the fractional portion is rounded.)

In the ON...GOSUB statement, each line number in the list must be the first line number of a subroutine.

If the value of <expression> is zero or greater than the number of items in the list (but less than or equal to 255), Microsoft BASIC continues with the next executable statement. If the value of <expression> is negative or greater than 255, an "Illegal function call" error occurs.

Example

```
100 ON L-1 GOTO 150,300,320,390
```

2.45 OPEN

Syntax OPEN <mode>,[#]<file number>,<filename>[,<reclen>]

Purpose To allow I/O to a disk file.

Remarks A disk file must be OPENed before any disk I/O operation can be performed on that file. OPEN allocates a buffer for I/O to the file and determines the mode of access that will be used with the buffer.

<mode> is a string expression whose first character is one of the following:

O Specifies sequential output mode.

I Specifies sequential input mode.

R Specifies random input/output mode.

<file number> is an integer expression whose value is between 1 and 15. The number is associated with the file as long as it is OPEN and refers other disk I/O statements to the file.

<filename> is a string expression containing a name that conforms to your operating system's rules for disk filenames.

<reclen> is an integer expression which, if included, sets the record length for random files. The default record length is 128 bytes.

Note A file can be OPENed for sequential input or random access on more than one file number at a time. A file may be OPENed for output, however, on only one file number at a time.

Example

```
10 OPEN "I",2,"INVEN"
```

See also "Disk File Handling," in the *Microsoft BASIC User's Guide.*

2.46 OPTION BASE

Syntax

```
OPTION BASE n
```

where n is 1 or 0

Purpose To declare the minimum value for array subscripts.

Remarks The default base is 0. If the statement

```
OPTION BASE 1
```

is executed, the lowest value an array subscript may have is 1.

Example

```
OPTION BASE 1
```

2.47 OUT

Syntax OUT I,J

where I and J are integer expressions in the range 0 to 255.

Purpose To send a byte to a machine output port.

Remarks The integer expression I is the port number. The integer expression J is the data to be transmitted.

Example

```
100 OUT 32,100
```

2.48 POKE

Syntax POKE I,J

where I and J are integer expressions.

Purpose To write a byte into a memory location.

Remarks I and J are integer expressions. The expression I represents the address of the memory location and J is the data byte. I must be in the range -32768 to 65535. (For interpretation of negative values of I, see "VARPTR," Section 3.43.)

The complementary function to POKE is PEEK. The argument to PEEK is an address from which a byte is to be read. See Section 3.28.

POKE and PEEK are useful for storing data efficiently, loading assembly language subroutines, and passing arguments and results to and from assembly language subroutines.

Example

```
10 POKE &H5A00,&HFF
```

2.49 PRINT

Syntax PRINT [<list of expressions>]

Purpose To output data at the terminal.

Remarks If <list of expressions> is omitted, a blank line is printed. If <list of expressions> is included, the values of the expressions are printed at the terminal. The expressions in the list may be numeric and/or string expressions. (Strings must be enclosed in quotation marks.)

Print Positions

The position of each printed item is determined by the punctuation used to separate the items in the list. Microsoft BASIC divides the line into print zones of 14 spaces each. In the list of expressions, a comma causes the next value to be printed at the beginning of the next zone. A semicolon causes the next value to be printed immediately after the last value. Typing one or more spaces between expressions has the same effect as typing a semicolon.

If a comma or a semicolon terminates the list of expressions, the next PRINT statement begins printing on the same line, spacing accordingly. If the list of expressions terminates without a comma or a semicolon, a carriage return is printed at the end of the line. If the printed line is longer than the terminal width, Microsoft BASIC goes to the next physical line and continues printing.

Printed numbers are always followed by a space. Positive numbers are preceded by a space. Negative numbers are preceded by a minus sign. Single precision numbers that can be represented with 6 or fewer digits in the unscaled format no less accurately than they can be represented in the scaled format are output using the unscaled format. For example, 1E-7 is output as .0000001 and 1E-8 is output as 1E-08. Double precision numbers that

can be represented with 16 or fewer digits in the unscaled format no less accurately than they can be represented in the scaled format are output using the unscaled format. For example, 1D-15 is output as .0000000000000001 and 1D-16 is output as 1D-16.

A question mark may be used in place of the word PRINT in a PRINT statement.

Example 1

```
10 X=5
20 PRINT X+5,X-5,X*(-5),X^5
30 END
RUN
 10            0            -25           3125
Ok
```

In this example, the commas in the PRINT statement cause each value to be printed at the beginning of the next print zone.

Example 2

```
LIST
10 INPUT X
20 PRINT X "SQUARED IS" X^2 "AND";
30 PRINT X "CUBED IS" X^3
40 PRINT
50 GOTO 10
Ok
RUN
? 9
 9 SQUARED IS 81 AND 9 CUBED IS 729

? 21
 21 SQUARED IS 441 AND 21 CUBED IS 9261

?
```

In this example, the semicolon at the end of line 20 causes both PRINT statements to be printed on the same line. Line 40 causes a blank line to be printed before the next prompt.

Example 3

```
10 FOR X=1 TO 5
20 J=J+5
30 K=K+10
40 ?J;K;
50 NEXT X
Ok
RUN
 5   10   10   20   15   30   20   40   25   50
Ok
```

In this example, the semicolons in the PRINT statement cause each value to be printed immediately after the preceding value. (Don't forget, a number is always followed by a space, and positive numbers are preceded by a space.) In line 40, a question mark is used instead of the word PRINT.

2.50 PRINT USING

Syntax

PRINT USING <string exp>;<list of expressions>

Purpose

To print strings or numbers using a specified format.

Remarks and Examples

<list of expressions> is comprised of the string expressions or numeric expressions that are to be printed, separated by semicolons. <string exp> is a string literal (or variable) comprised of special formatting characters. These formatting characters (see below) determine the field and the format of the printed strings or numbers.

String Fields

When PRINT USING is used to print strings, one of three formatting characters may be used to format the string field:

"!"

Specifies that only the first character in the given string is to be printed.

"\n spaces\" Specifies that 2+n characters from the string are to be printed. If the backslashes are typed with no spaces, two characters will be printed; with one space, three characters will be printed, and so on. If the string is longer than the field, the extra characters are ignored. If the field is longer than the string, the string will be left-justified in the field and padded with spaces on the right.

Example

```
10 A$ = "LOOK":B$ = "OUT"
30 PRINT USING "!";A$;B$
40 PRINT USING "\ \";A$;B$
50 PRINT USING "\ \";A$;B$;"!!"
RUN
LO
LOOKOUT
LOOK OUT !!
```

"&" Specifies a variable length string field. When the field is specified with "&", the string is output without modification.

Example

```
10 A$ = "LOOK":B$ = "OUT"
20 PRINT USING "!";A$;
30 PRINT USING "&";B$
RUN
LOUT
```

Numeric Fields

When PRINT USING is used to print numbers, the following special characters may be used to format the numeric field:

A number sign is used to represent each digit position. Digit positions are always filled. If the number to be printed has fewer digits than positions specified, the number will be right-justified (preceded by spaces) in the field.

A decimal point may be inserted at any position in the field. If the format string specifies that a digit is to precede the decimal point, the digit will always be printed (as 0, if necessary). Numbers are rounded as necessary.

```
PRINT USING "##.##";.78
  0.78

PRINT USING "###.##";987.654
987.65

PRINT USING "##.##  ";10.2,5.3,66.789,.234
10.20  5.30  66.79  0.23
```

In the last example, three spaces were inserted at the end of the format string to separate the printed values on the line.

+ A plus sign at the beginning or end of the format string will cause the sign of the number (plus or minus) to be printed before or after the number.

- A minus sign at the end of the format field will cause negative numbers to be printed with a trailing minus sign.

```
PRINT USING "+##.##  ";-68.95,2.4,55.6,-.9
-68.95  +2.40  +55.60  -0.90

PRINT USING "##.##-  ";-68.95,22.449,-7.01
68.95-  22.45  7.01-
```

****** A double asterisk at the beginning of the format string causes leading spaces in the numeric field to be filled with asterisks. The ****** also specifies positions for two more digits.

```
PRINT USING "**#.#  ";12.39,-0.9,765.1
*12.4  *-0.9  765.1
```

$$

A double dollar sign causes a dollar sign to be printed to the immediate left of the formatted number. The $$ specifies two more digit positions, one of which is the dollar sign. The exponential format cannot be used with $$. Negative numbers cannot be used unless the minus sign trails to the right.

```
PRINT USING "$$###.##";456.78
  $456.78
```

****$**

The **$ at the beginning of a format string combines the effects of the above two symbols. Leading spaces will be asterisk-filled and a dollar sign will be printed before the number. **$ specifies three more digit positions, one of which is the dollar sign.

```
PRINT USING "**$##.##";2.34
***$2.34
```

,

A comma that is to the left of the decimal point in a formatting string causes a comma to be printed to the left of every third digit to the left of the decimal point. A comma that is at the end of the format string is printed as part of the string. A comma specifies another digit position. The comma has no effect if used with the exponential (^^^^) format.

```
PRINT USING "####,.##";1234.5
1,234.50

PRINT USING "####.##,";1234.5
1234.50,
```

^^^^

Four carets (or up-arrows) may be placed after the digit position characters to specify exponential format. The four carets allow space for E+xx to be printed. Any decimal point position may be specified. The significant digits are left-justified, and the exponent is adjusted. Unless a leading + or trailing + or - is specified, one digit position will be used to the left of the decimal point to print a space or a minus sign.

```
PRINT USING "##.##^^^^";234.56
 2.35E+02

PRINT USING ".####^^^^-";888888
.8889E+06

PRINT USING "+.##^^^^";123
+.12E+03
```

_ An underscore in the format string causes the next character to be output as a literal character.

```
PRINT USING "_!##.##_!";12.34
!12.34!
```

The literal character itself may be an underscore by placing "__" in the format string.

% If the number to be printed is larger than the specified numeric field, a percent sign is printed in front of the number. If rounding causes the number to exceed the field, a percent sign will be printed in front of the rounded number.

```
PRINT USING "##.##";111.22
%111.22

PRINT USING ".##";.999
%1.00
```

If the number of digits specified exceeds 24, an "Illegal function call" error will result.

2.51 PRINT# and PRINT# USING

Syntax PRINT#<file number>,[USING <string exp>;] <list of expressions>

Purpose To write data to a sequential disk file.

Remarks <file number> is the number that was used when

the file was OPENed for output. <string exp> is comprised of formatting characters as described in Section 2.50, "PRINT USING." The expressions in <list of expressions> are the numeric and/or string expressions that will be written to the file.

PRINT# does not compress data on the disk. An image of the data is written to the disk, just as it would be displayed on the terminal screen with a PRINT statement. For this reason, care should be taken to delimit the data on the disk, so that it will be input correctly from the disk.

In the list of expressions, numeric expressions should be delimited by semicolons. For example:

```
PRINT#1,A;B;C;X;Y;Z
```

(If commas are used as delimiters, the extra blanks that are inserted between print fields will also be written to the disk.)

String expressions must be separated by semicolons in the list. To format the string expressions correctly on the disk, use explicit delimiters in the list of expressions.

For example, let A$="CAMERA" and B$= "93604-1". The statement

```
PRINT#1,A$;B$
```

would write CAMERA93604-1 to the disk. Because there are no delimiters, this could not be input as two separate strings. To correct the problem, insert explicit delimiters into the PRINT# statement as follows:

```
PRINT#1,A$;",";B$
```

The image written to disk is

```
CAMERA,93604-1
```

which can be read back into two string variables.

If the strings themselves contain commas, semicolons, significant leading blanks, carriage returns, or line feeds, write them to disk surrounded by explicit quotation marks, CHR$(34).

For example, let A$="CAMERA, AUTOMATIC" and B$=" 93604-1". The statement

```
PRINT#1,A$;B$
```

would write the following image to disk:

```
CAMERA, AUTOMATIC  93604-1
```

And the statement

```
INPUT#1,A$,B$
```

would input "CAMERA" to A$ and "AUTOMATIC 93604-1" to B$. To separate these strings properly on the disk, write double quotation marks to the disk image using CHR$(34). The statement

```
PRINT#1,CHR$(34);A$;CHR$(34);CHR$(34);B$;
CHR$(34)
```

writes the following image to disk:

```
"CAMERA, AUTOMATIC""  93604-1"
```

And the statement

```
INPUT#1,A$,B$
```

would input "CAMERA, AUTOMATIC" to A$ and " 93604-1" to B$.

The PRINT# statement may also be used with the USING option to control the format of the disk file. For example:

```
PRINT#1,USING"$$###.##,";J;K;L
```

For more examples using PRINT#, see "Disk File Handling," in the *Microsoft BASIC User's Guide.*

See also "WRITE#," Section 2.68.

2.52 PUT

Syntax PUT [#]<file number>[,<record number>]

Purpose To write a record from a random buffer to a random disk file.

Remarks <file number> is the number under which the file was OPENed. If <record number> is omitted, the record will assume the next available record number (after the last PUT). The largest possible record number is 32,767. The smallest record number is 1.

Example See "Disk File Handling," in the *Microsoft BASIC User's Guide.*

Note PRINT#, PRINT# USING, and WRITE# may be used to put characters in the random file buffer before executing a PUT statement.

In the case of WRITE#, Microsoft BASIC pads the buffer with spaces up to the carriage return. Any attempt to read or write past the end of the buffer causes a "Field overflow" error.

2.53 RANDOMIZE

Syntax RANDOMIZE [<expression>]

Purpose To reseed the random number generator.

Remarks If <expression> is omitted, Microsoft BASIC suspends program execution and asks for a value by printing

Random Number Seed (-32768 to 32767)?

before executing RANDOMIZE.

If the random number generator is not reseeded, the RND function returns the same sequence of random numbers each time the program is RUN. To change the sequence of random numbers every time the program is RUN, place a RANDOMIZE statement at the beginning of the program and change the argument with each RUN.

Example

```
10 RANDOMIZE
20 FOR I = 1 TO 5
30 PRINT RND;
40 NEXT I
RUN
Random Number Seed (-32768 to 32767)? 3
     (user types 3)
 .88598 .484668 .586328 .119426 .709225
Ok
RUN
Random Number Seed (-32768 to 32767)? 4
     (user types 4 for new sequence)
 .803506 .162462 .929364 .292443 .322921
Ok
RUN
Random Number Seed (-32768 to 32767)? 3
     (same sequence as first RUN)
 .88598 .484668 .586328 .119426 .709225
Ok
```

2.54 READ

Syntax READ <list of variables>

Purpose To read values from a DATA statement and assign them to variables. (See "DATA," Section 2.10.)

Remarks A READ statement must always be used in conjunction with a DATA statement. READ statements assign variables to DATA statement values on a one-to-one basis. READ statement variables may be numeric or string, and the values read must agree with the variable types specified. If they do not agree, a "Syntax error" will result.

A single READ statement may access one or more DATA statements (they will be accessed in order), or several READ statements may access the same DATA statement. If the number of variables in <list of variables> exceeds the number of elements in the DATA statement(s), an "Out of data" error message is printed. If the number of variables specified is fewer than the number of elements in the DATA statement(s), subsequent READ statements will begin reading data at the first unread element. If there are no subsequent READ statements, the extra data is ignored.

To reread DATA statements from the start, use the RESTORE statement (see "RESTORE," Section 2.57).

Example 1

```
.
.
.
80 FOR I=1 TO 10
90 READ A(I)
100 NEXT I
110 DATA 3.08,5.19,3.12,3.98,4.24
120 DATA 5.08,5.55,4.00,3.16,3.37
.
.
.
```

This program segment READs the values from the DATA statements into the array A. After execution, the value of A(1) will be 3.08, and so on.

Example 2

```
LIST
10 PRINT "CITY", "STATE", " ZIP"
20 READ C$,S$,Z
30 DATA "DENVER,", COLORADO, 80211
40 PRINT C$,S$,Z
Ok
RUN
CITY            STATE           ZIP
DENVER,         COLORADO        80211
Ok
```

This program READs string and numeric data from the DATA statement in line 30.

2.55 REM

Syntax

REM <remark>

Purpose

To allow explanatory remarks to be inserted in a program.

Remarks

REM statements are not executed but are output exactly as entered when the program is listed.

REM statements may be branched into from a GOTO or GOSUB statement. Execution will continue with the first executable statement after the REM statement.

Remarks may be added to the end of a line by preceding the remark with a single quotation mark instead of :REM.

Important

Do not use this in a data statement, because it would be considered legal data.

Example

```
120 REM CALCULATE AVERAGE VELOCITY
130 FOR I = 1 TO 20
140 SUM = SUM + V(I)
```

or

```
120 FOR I = 1 TO 20      'CALCULATE AVERAGE
VELOCITY
130 SUM = SUM + V(I)
140 NEXT I
```

2.56 RENUM

Syntax RENUM [[<new number>][,[<old number>] [,<increment>]]]

Purpose To renumber program lines.

Remarks <new number> is the first line number to be used in the new sequence. The default is 10. <old number> is the line in the current program where renumbering is to begin. The default is the first line of the program. <increment> is the increment to be used in the new sequence. The default is 10.

RENUM also changes all line number references following GOTO, GOSUB, THEN, ON...GOTO, ON...GOSUB, and ERL statements to reflect the new line numbers. If a nonexistent line number appears after one of these statements, the error message "Undefined line number in xxxxx" is

printed. The incorrect line number reference is not changed by RENUM, but line number yyyyy may be changed.

Note

RENUM cannot be used to change the order of program lines (for example, RENUM 15,30 when the program has three lines numbered 10, 20, and 30) or to create line numbers greater than 65529. An "Illegal function call" error will result.

Examples

RENUM	Renumbers the entire program. The first new line number will be 10. Lines will be numbered in increments of 10.
RENUM 300,,50	Renumbers the entire program. The first new line number will be 300. Lines will be numbered in increments of 50.
RENUM 1000,900,20	Renumbers the lines from 900 up so they start with line number 1000 and are numbered in increments of 20.

2.57 RESTORE

Syntax

RESTORE [<line number>]

Purpose

To allow DATA statements to be reread from a specified line.

Remarks

After a RESTORE statement is executed, the next READ statement accesses the first item in the first DATA statement in the program. If <line number> is specified, the next READ statement accesses the first item in the specified DATA statement.

Example

```
10 READ A,B,C
20 RESTORE
30 READ D,E,F
40 DATA 57, 68, 79
.
.
.
```

2.58 RESUME

Syntax

RESUME

RESUME 0

RESUME NEXT

RESUME <line number>

Purpose

To continue program execution after an error recovery procedure has been performed.

Remarks

Any one of the four syntax shown above may be used, depending upon where execution is to resume:

RESUME or RESUME 0	Execution resumes at the statement which caused the error.
RESUME NEXT	Execution resumes at the statement immediately following the one which caused the error.
RESUME <line number>	Execution resumes at <line number>.

A RESUME statement that is not in an error handling routine causes a "RESUME without error" message to be printed.

Example

```
10 ON ERROR GOTO 900
.
.
.
900 IF (ERR = 230)AND(ERL = 90) THEN PRINT
"TRY AGAIN":RESUME 80
.
.
.
```

2.59 RUN

Syntax 1 RUN [<line number>]

Purpose To execute the program currently in memory.

Remarks If <line number> is specified, execution begins on that line. Otherwise, execution begins at the lowest line number. Microsoft BASIC always returns to command level after a RUN is executed.

Example

```
RUN
```

Syntax 2 RUN <filename>[,R]

Purpose To load a file from disk into memory and run it.

Remarks <filename> is the name used when the file was SAVEd. (Your operating system may append a default filename extension if one was not supplied in the SAVE command. Refer to "Disk File Handling," in the *Microsoft BASIC User's Guide* for information about possible filename extensions under your operating system.)

RUN closes all open files and deletes the current contents of memory before loading the designated program. However, with the R option, all data files remain OPEN.

Example

```
RUN "NEWFIL",R
```

See also "Microsoft BASIC Disk I/O," in the *Microsoft BASIC User's Guide.*

Note Microsoft BASIC Compiler supports the RUN and RUN <line number> forms of the RUN statement. Microsoft BASIC Compiler does not support the R option with RUN. If you want this feature, the CHAIN statement should be used.

2.60 SAVE

Syntax SAVE <filename>[{,A|,P}]

Purpose To save a program file on disk.

Remarks <filename> is a quoted string that conforms to your operating system's requirements for filenames. (Your operating system may append a default filename extension if one was not supplied in the SAVE command. Refer to "Disk File Handling," in the *Microsoft BASIC User's Guide* for information about possible filename extensions under your operating system.) If <filename> already exists, the file will be written over.

Use the A option to save the file in ASCII format. Otherwise, Microsoft BASIC saves the file in a compressed binary format. ASCII format takes more space on the disk, but some disk access requires that files be in ASCII format. For instance, the MERGE command requires an ASCII format file, and some operating system commands such as LIST may require an ASCII format file.

Use the P option to protect the file by saving it in an encoded binary format. When a protected file is later RUN (or LOADed), any attempt to list or edit it will fail.

Examples

```
SAVE "COM2",A
SAVE "PROG",P
```

See also "Disk File Handling," in the *Microsoft BASIC User's Guide.*

2.61 STOP

Syntax STOP

Purpose To terminate program execution and return to command level.

Remarks STOP statements may be used anywhere in a program to terminate execution. When a STOP is encountered, the following message is printed:

```
Break in line nnnnn
```

Unlike the END statement, the STOP statement does not close files.

Microsoft BASIC always returns to command level after a STOP is executed. Execution is resumed by issuing a CONT command (see Section 2.8).

Example

```
10 INPUT A,B,C
20 K=A^2*5.3:L=B^3/.26
30 STOP
40 M=C*K+100:PRINT M
RUN
? 1,2,3
BREAK IN 30
Ok
PRINT L
  30.7692
Ok
CONT
  115.9
Ok
```

2.62 SWAP

Syntax SWAP <variable>,<variable>

Purpose To exchange the values of two variables.

Remarks Any type variable may be SWAPped (integer, single precision, double precision, string), but the two variables must be of the same type or a "Type mismatch" error results.

Example

```
LIST
10 A$=" ONE " : B$=" ALL " : C$="FOR"
20 PRINT A$ C$ B$
30 SWAP A$, B$
40 PRINT A$ C$ B$
RUN
Ok
  ONE FOR ALL
  ALL FOR ONE
Ok
```

2.63 TRON/TROFF

Syntax TRON

TROFF

Purpose To trace the execution of program statements.

Remarks As an aid in debugging, the TRON statement (executed in either direct or indirect mode) enables a trace flag that prints each line number of the program as it is executed. The numbers appear enclosed in square brackets. The trace flag is disabled with the TROFF statement (or when a NEW command is executed).

Example

```
TRON
Ok
LIST
10 K=10
20 FOR J=1 TO 2
30 L=K + 10
40 PRINT J;K;L
50 K=K+10
60 NEXT
70 END
Ok
RUN
[10][20][30][40] 1 10 20
[50][60][30][40] 2 20 30
[50][60][70]
Ok
TROFF
Ok
```

2.64 WAIT

Syntax

WAIT <port number>,I[,J]

where I and J are integer expressions.

Purpose

To suspend program execution while monitoring the status of a machine input port.

Remarks

The WAIT statement causes execution to be suspended until a specified machine input port develops a specified bit pattern. The data read at the port is exclusive OR'ed with the integer expression J, and then AND'ed with I. If the result is zero, Microsoft BASIC loops back and reads the data at the port again. If the result is nonzero, execution continues with the next statement. If J is omitted, it is assumed to be zero.

Important

It is possible to enter an infinite loop with the WAIT statement, in which case it will be necessary to manually restart the machine. To avoid this, WAIT must have the specified value at

<port number> during some point in the program execution.

Example

```
100 WAIT 32,2
```

2.65 WHILE...WEND

Syntax

```
WHILE <expression>
.
.
.
[<loop statements>]
.
.
WEND
```

Purpose

To execute a series of statements in a loop as long as a given condition is true.

Remarks

If <expression> is not zero (i.e., true), <loop statements> are executed until the WEND statement is encountered. Microsoft BASIC then returns to the WHILE statement and checks <expression>. If it is still true, the process is repeated. If it is not true, execution resumes with the statement following the WEND statement.

WHILE/WEND loops may be nested to any level. Each WEND will match the most recent WHILE. An unmatched WHILE statement causes a "WHILE without WEND" error, and an unmatched WEND statement causes a "WEND without WHILE" error.

Example

```
90 'BUBBLE SORT ARRAY A$
100 FLIPS = 1 'FORCE ONE PASS THRU LOOP
110 WHILE FLIPS
115         FLIPS = 0
120         FOR I = 1 TO J-1
130             IF A$(I)>A$(I + 1) THEN
                        SWAP A$(I),A$(I + 1):FLIPS = 1
140         NEXT I
150 WEND
```

2.66 WIDTH

Syntax

WIDTH [LPRINT]<integer expression>

Purpose

To set the printed line width in number of characters for the terminal or line printer.

Remarks

If the LPRINT option is omitted, the line width is set at the terminal. If LPRINT is included, the line width is set at the line printer.

<integer expression> must have a value in the range 15 to 255. The default width is 72 characters.

If <integer expression> is 255, the line width is "infinite"; that is, Microsoft BASIC never inserts a carriage return. However, the position of the cursor or the print head, as given by the POS or LPOS function, returns to zero after position 255.

Example

```
10 PRINT "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
RUN
ABCDEFGHIJKLMNOPQRSTUVWXYZ
Ok
WIDTH 18
Ok
RUN
ABCDEFGHIJKLMNOPQR
STUVWXYZ
Ok
```

2.67 WRITE

Syntax WRITE [<list of expressions>]

Purpose To output data at the terminal.

Remarks If <list of expressions> is omitted, a blank line is output. If <list of expressions> is included, the values of the expressions are output at the terminal. The expressions in the list may be numeric and/or string expressions. They must be separated by commas.

When the printed items are output, each item is separated from the last by a comma. Printed strings are delimited by quotation marks. After the last item in the list is printed, Microsoft BASIC inserts a carriage return/line feed.

WRITE outputs numeric values using the same syntax as the PRINT statement. (See Section 2.49.)

Example

```
10 A=80:B=90:C$="THAT'S ALL"
20 WRITE A,B,C$
RUN
  80, 90,"THAT'S ALL"
Ok
```

2.68 WRITE#

Syntax WRITE#<file number>,<list of expressions>

Purpose To write data to a sequential file.

Remarks <file number> is the number under which the file was OPENed in "O" mode (see "OPEN," Section 2.45). The expressions in the list are string or numeric expressions. They must be separated by commas.

The difference between WRITE# and PRINT# is that WRITE# inserts commas between the items as they are written to disk and delimits strings with quotation marks. Therefore, it is not necessary for the user to put explicit delimiters in the list. A carriage return/line feed sequence is inserted after the last item in the list is written to disk.

Example Let A$ = "CAMERA" and B$ = "93604-1"

The statement

```
WRITE#1,A$,B$
```

writes the following image to disk:

```
"CAMERA","93604-1"
```

A subsequent INPUT# statement, such as

```
INPUT#1,A$,B$
```

would input "CAMERA" to A$ and "93604-1" to B$.

Chapter 3

Microsoft BASIC Functions

Introduction 107

3.1 ABS 107

3.2 ASC 108

3.3 ATN 108

3.4 CDBL 109

3.5 CHR$ 109

3.6 CINT 110

3.7 COS 110

3.8 CSNG 111

3.9 CVI, CVS, CVD 111

3.10 EOF 112

3.11 EXP 112

3.12 FIX 113

3.13 FRE 113

3.14 HEX$ 114

3.15 INKEY$ 114

3.16 INP 115

3.17 INPUT$ 115

3.18 INSTR 116

3.19 INT 116

3.20 LEFT$ 117

3.21 LEN 117
3.22 LOC 117
3.23 LOG 118
3.24 LPOS 118
3.25 MID$ 119
3.26 MKI$, MKS$, MKD$ 119
3.27 OCT$ 120
3.28 PEEK 120
3.29 POS 121
3.30 RIGHT$ 121
3.31 RND 122
3.32 SGN 122
3.33 SIN 123
3.34 SPACE$ 123
3.35 SPC 124
3.36 SQR 124
3.37 STR$ 125
3.38 STRING$ 125
3.39 TAB 125
3.40 TAN 126
3.41 USR 126
3.42 VAL 127
3.43 VARPTR 128

Microsoft BASIC intrinsic functions are described in this chapter. The functions may be called from any program without further definition.

Arguments to functions are always enclosed in parentheses. In the syntaxes given for the functions in this chapter, the arguments have been abbreviated as follows:

X and Y	Represent any numeric expressions.
I and J	Represent integer expressions.
X$ and Y$	Represent string expressions.

If a floating-point value is supplied where an integer is required, Microsoft BASIC will round the fractional portion and use the resulting integer.

Note

With Microsoft BASIC Interpreter, only integer and single precision results are returned by functions. Double precision functions are supported only by the Microsoft BASIC Compiler.

3.1 ABS

Syntax ABS(X)

Action Returns the absolute value of the expression X.

Example

```
PRINT ABS(7*(-5))
 35
Ok
```

3.2 ASC

Syntax ASC(X$)

Action Returns a numerical value that is the ASCII code for the first character of the string X$. (See Appendix C for ASCII codes.) If X$ is null, an "Illegal function call" error is returned.

Example

```
10 X$ = "TEST"
20 PRINT ASC(X$)
RUN
 84
Ok
```

See the CHR$ function, Section 3.5, for details on ASCII-to-string conversion.

3.3 ATN

Syntax ATN(X)

Action Returns the arctangent of X in radians. Result is in the range -pi/2 to pi/2. The expression X may be any numeric type, but the evaluation of ATN is always performed in single precision.

Example

```
10 INPUT X
20 PRINT ATN(X)
RUN
? 3
 1.24905
Ok
```

3.4 CDBL

Syntax CDBL(X)

Action Converts X to a double precision number.

Example

```
10 A = 454.67
20 PRINT A;CDBL(A)
RUN
 454.67 454.6700134277344
Ok
```

3.5 CHR$

Syntax CHR$(I)

Action Returns a string whose one character is ASCII character I. (ASCII codes are listed in Appendix C.) CHR$ is commonly used to send a special character to the terminal. For instance, the BEL character (CHR$(7)) could be sent as a preface to an error message, or a form feed (CHR$(12)) could be sent to clear a terminal screen and return the cursor to the home position.

Example

```
PRINT CHR$(66)
B
Ok
```

See the ASC function, Section 3.2, for details on ASCII-to-numeric conversion.

3.6 CINT

Syntax CINT(X)

Action Converts X to an integer by rounding the fractional portion. If X is not in the range -32768 to 32767, an "Overflow" error occurs.

Example

```
PRINT CINT(45.67)
 46
Ok
```

See the CDBL and CSNG functions for details on converting numbers to the double precision and single precision data type, respectively. See also the FIX and INT functions, both of which return integers.

3.7 COS

Syntax COS(X)

Action Returns the cosine of X in radians. The calculation of COS(X) is performed in single precision.

Example

```
10 X = 2*COS(.4)
20 PRINT X
RUN
 1.84212
Ok
```

3.8 CSNG

Syntax CSNG(X)

Action Converts X to a single precision number.

Example

```
10 A# = 975.3421#
20 PRINT A#; CSNG(A#)
RUN
 975.3421 975.342
Ok
```

See the CINT and CDBL functions for converting numbers to the integer and double precision data types, respectively.

3.9 CVI, CVS, CVD

Syntax

```
CVI(<2-byte string>)
CVS(<4-byte string>)
CVD(<8-byte string>)
```

Action Convert string values to numeric values. Numeric values that are read in from a random disk file must be converted from strings back into numbers. CVI converts a 2-byte string to an integer. CVS converts a 4-byte string to a single precision number. CVD converts an 8-byte string to a double precision number.

Example

```
.
.
.
70 FIELD #1,4 AS N$, 12 AS B$, ...
80 GET #1
90 Y = CVS(N$)
.
.
.
```

See also "MKI$, MKS$, MKD$," Section 3.26 and "Disk File Handling," in the *Microsoft BASIC User's Guide.*

3.10 EOF

Syntax EOF(<file number>)

Action Returns -1 (true) if the end of a sequential file has been reached. Use EOF to test for end-of-file while INPUTting, to avoid "Input past end" errors.

Example

```
10 OPEN "I",1,"DATA"
20 C=0
30 IF EOF(1) THEN 100
40 INPUT #1,M(C)
50 C=C+1:GOTO 30
 .
 .
 .
```

3.11 EXP

Syntax EXP(X)

Action Returns e (base of natural logarithms) to the power of X. X must be <=87.3365. If EXP overflows, the "Overflow" error message is displayed, machine infinity with the appropriate sign is supplied as the result, and execution continues.

Example

```
10 X=5
20 PRINT EXP(X-1)
RUN
 54.5982
Ok
```

3.12 FIX

Syntax FIX(X)

Action Returns the truncated integer part of X. FIX(X) is equivalent to SGN(X)*INT(ABS(X)). The major difference between FIX and INT is that FIX does not return the next lower number for negative X.

Examples

```
PRINT FIX(58.75)
 58
Ok

PRINT FIX(-58.75)
-58
Ok
```

3.13 FRE

Syntax

```
FRE(0)
FRE(" ")
```

Action Arguments to FRE are dummy arguments. FRE returns the number of bytes in memory not being used by Microsoft BASIC.

FRE(" ") forces a garbage collection before returning the number of free bytes. Be patient: garbage collection may take 1 to 1½ minutes.

Microsoft BASIC will not initiate garbage collection until all free memory has been used up. Therefore, using FRE(" ") periodically will result in shorter delays for each garbage collection.

Example

```
PRINT FRE(0)
 14542
Ok
```

3.14 HEX$

Syntax HEX$(X)

Action Returns a string which represents the hexadecimal value of the decimal argument. X is rounded to an integer before HEX$(X) is evaluated.

Example

```
10 INPUT X
20 A$ = HEX$(X)
30 PRINT X "DECIMAL IS" A$ "HEXADECIMAL"
RUN
? 32
 32 DECIMAL IS 20 HEXADECIMAL
Ok
```

See the OCT$ function, Section 3.27, for details on octal conversion.

3.15 INKEY$

Syntax INKEY$

Action Returns either a one-character string containing a character read from the terminal or a null string if no character is pending at the terminal. No characters will be echoed. All characters are passed through to the program except for Control-C, which terminates the program. (With Microsoft BASIC Compiler, Control-C is also passed through to the program.)

Example

```
1000 'TIMED INPUT SUBROUTINE
1010 RESPONSE$ = " "
1020 FOR I% = 1 TO TIMELIMIT%
1030 A$ = INKEY$ : IF LEN(A$) = 0 THEN 1060
1040 IF ASC(A$) = 13 THEN TIMEOUT% = 0 : RETURN
1050 RESPONSE$ = RESPONSE$ + A$
1060 NEXT I%
1070 TIMEOUT% = 1 : RETURN
```

3.16 INP

Syntax INP(I)

Action Returns the byte read from port I. I must be in the range 0 to 255. INP is the complementary function to the OUT statement, Section 2.47.

Example

```
100 A = INP(255)
```

3.17 INPUT$

Syntax INPUT$(X[,[#]Y])

Action Returns a string of X characters, read from the terminal or from file number Y. If the terminal is used for input, no characters will be echoed. All control characters are passed through except Control-C, which is used to interrupt the execution of the INPUT$ function.

Example 1

```
5 'LIST THE CONTENTS OF A SEQUENTIAL FILE
IN HEXADECIMAL
10 OPEN"I",1,"DATA"
20 IF EOF(1) THEN 50
30 PRINT HEX$(ASC(INPUT$(1,#1)));
40 GOTO 20
50 PRINT
60 END
```

Example 2

```
.
.
.
100 PRINT "TYPE P TO PROCEED OR S TO STOP"
110 X$ = INPUT$(1)
120 IF X$ = "P" THEN 500
130 IF X$ = "S" THEN 700 ELSE 100
.
.
.
```

3.18 INSTR

Syntax INSTR([I,]X$,Y$)

Action Searches for the first occurrence of string Y$ in X$, and returns the position at which the match is found. Optional offset I sets the position for starting the search. I must be in the range 1 to 255. If I is greater than the number of characters in X$ (LEN(X$)), or if X$ is null, or Y$ cannot be found, INSTR returns 0. If Y$ is null, INSTR returns I or 1. X$ and Y$ may be string variables, string expressions, or string literals.

Example

```
10 X$ = "ABCDEB"
20 Y$ = "B"
30 PRINT INSTR(X$,Y$);INSTR(4,X$,Y$)
RUN
 2  6
Ok
```

Note If I=0 is specified, the "Illegal function call" error message will be returned.

3.19 INT

Syntax INT(X)

Action Returns the largest integer <=X.

Examples

```
PRINT INT(99.89)
 99
Ok

PRINT INT(-12.11)
-13
Ok
```

See the CINT and FIX functions, Sections 3.6 and 3.12, respectively, which also return integer values.

3.20 LEFT$

Syntax LEFT$(X$,I)

Action Returns a string comprised of the leftmost I characters of X$. I must be in the range 0 to 255. If I is greater than the number of characters in X$ (LEN(X$)), the entire string (X$) will be returned. If I=0, the null string (length zero) is returned.

Example

```
10 A$ = "BASIC"
20 B$ = LEFT$(A$,5)
30 PRINT B$
BASIC
Ok
```

Also see the MID$ and RIGHT$ functions, Sections 3.25 and 3.30, respectively.

3.21 LEN

Syntax LEN(X$)

Action Returns the number of characters in X$. Nonprinting characters and blanks are counted.

Example

```
10 X$ = "PORTLAND, OREGON"
20 PRINT LEN(X$)
 16
Ok
```

3.22 LOC

Syntax LOC(<file number>)

where <file number> is the number under which the file was OPENed.

Action With random disk files, LOC returns the record number just read or written from a GET or PUT statement. If the file was opened but no disk I/O has been performed yet, LOC returns a 0. With sequential files, LOC returns the number of sectors (128-byte blocks) read from or written to the file since it was OPENed.

Example

```
200 IF LOC(1)>50 THEN STOP
```

3.23 LOG

Syntax LOG(X)

Action Returns the natural logarithm of X. X must be greater than zero.

Example

```
PRINT LOG(45/7)
 1.86075
Ok
```

3.24 LPOS

Syntax LPOS(X)

Action Returns the current position of the line printer print head within the line printer's buffer. Does not necessarily give the physical position of the print head. X is a dummy argument.

Example

```
100 IF LPOS(X)>60 THEN LPRINT CHR$(13)
```

3.25 MID$

Syntax MID$(X$,I[,J])

Action Returns a string of length J characters from X$, beginning with the Ith character. I and J must be in the range 1 to 255. If J is omitted or if there are fewer than J characters to the right of the Ith character, all rightmost characters beginning with the Ith character are returned. If I is greater than the number of characters in X$ (LEN(X$)), MID$ returns a null string.

Example

```
LIST
10 A$ = "GOOD "
20 B$ = "MORNING EVENING AFTERNOON"
30 PRINT A$;MID$(B$,9,7)
Ok
RUN
GOOD EVENING
Ok
```

Also see the LEFT$ and RIGHT$ functions, Sections 3.20 and 3.30, respectively.

If I=0 is specified, the "Illegal function call" error message will be returned.

3.26 MKI$, MKS$, MKD$

Syntax

```
MKI$(<integer expression>)
MKS$(<single precision expression>)
MKD$(<double precision expression>)
```

Action Convert numeric values to string values. Any numeric value that is placed in a random file buffer with an LSET or RSET statement must be converted to a string. MKI$ converts an integer to a 2-byte string. MKS$ converts a single precision number to a 4-byte string. MKD$ converts a double precision number to an 8-byte string.

Example

```
90 AMT = (K + T)
100 FIELD #1,8 AS D$,20 AS N$
110 LSET D$ = MKS$(AMT)
120 LSET N$ = A$
130 PUT #1
 .
 .
```

Also see "CVI, CVS, CVD," Section 3.9, and "Disk File Handling," in the *Microsoft BASIC User's Guide.*

3.27 OCT$

Syntax OCT$(X)

Action Returns a string which represents the octal value of the decimal argument. X is rounded to an integer before OCT$(X) is evaluated.

Example

```
PRINT OCT$(24)
 30
Ok
```

See the HEX$ function, Section 3.14, for details on hexadecimal conversion.

3.28 PEEK

Syntax PEEK(I)

Action Returns the byte read from the indicated memory location (I).

Remarks The returned value is an integer in the range 0 to 255. I must be in the range -32768 to 65535. (For the interpretation of a negative value of I, see "VARPTR," Section 3.43.)

PEEK is the complementary function of the POKE statement.

Example

```
A = PEEK(&H5A00)
```

3.29 POS

Syntax

```
POS(I)
```

Action

Returns the current cursor position. The leftmost position is 1. X is a dummy argument.

Example

```
IF POS(X)>60 THEN PRINT CHR$(13)
```

Also see the LPOS function, Section 3.24.

3.30 RIGHT$

Syntax

```
RIGHT$(X$,I)
```

Action

Returns the rightmost I characters of string X$. If I is equal to the number of characters in X$ (LEN(X$)), RIGHT$ returns X$. If I=0, the null string (length zero) is returned.

Example

```
10 A$ = "DISK BASIC"
20 PRINT RIGHT$(A$,5)
RUN
BASIC
Ok
```

Also see the LEFT$ and MID$ functions, Sections 3.20 and 3.25, respectively.

3.31 RND

Syntax RND[(X)]

Action Returns a random number between 0 and 1. The same sequence of random numbers is generated each time the program is RUN unless the random number generator is reseeded (see "RANDOMIZE," Section 2.53). However, X<0 always restarts the same sequence for any given X.

X>0 or X omitted generates the next random number in the sequence. X=0 repeats the last number generated.

Example

```
10 FOR I=1 TO 5
20 PRINT INT(RND*100);
30 NEXT
RUN
  24 30 31 51 5
Ok
```

Note The values produced by the RND function may vary with different implementations of Microsoft BASIC.

3.32 SGN

Syntax SGN(X)

Action If X>0, SGN(X) returns 1.
If X=0, SGN(X) returns 0.
If X<0, SGN(X) returns -1.

Example ON SGN(X)+2 GOTO 100,200,300

branches to 100 if X is negative, 200 if X is 0, and 300 if X is positive.

3.33 SIN

Syntax SIN(X)

Action Returns the sine of X in radians. SIN(X) is calculated in single precision. COS(X)=SIN(X+3.14159/2).

Example

```
PRINT SIN(1.5)
 .997495
Ok
```

Also see the COS(X) function, Section 3.7.

3.34 SPACE$

Syntax SPACE$(X)

Action Returns a string of spaces of length X. The expression X is rounded to an integer and must be in the range 0 to 255.

Example

```
10 FOR I = 1 TO 5
20 X$ = SPACE$(I)
30 PRINT X$;I
40 NEXT I
RUN
  1
    2
      3
        4
          5
Ok
```

Also see the SPC function, Section 3.35.

3.35 SPC

Syntax SPC(I)

Action Prints I blanks on the terminal. SPC may only be used with PRINT and LPRINT statements. I must be in the range 0 to 255. A";" is assumed to follow the SPC(I) command.

Example

```
PRINT "OVER" SPC(15) "THERE"
OVER                THERE
Ok
```

Also see the SPACE$ function, Section 3.34.

3.36 SQR

Syntax SQR(X)

Action Returns the square root of X. X must be >=0.

Example

```
10 FOR X=10 TO 25 STEP 5
20 PRINT X, SQR(X)
30 NEXT
RUN
  10          3.16228
  15          3.87298
  20          4.47214
  25          5
Ok
```

3.37 STR$

Syntax STR$(X)

Action Returns a string representation of the value of X.

Example

```
5 REM ARITHMETIC FOR KIDS
10 INPUT "TYPE A NUMBER";N
20 ON LEN(STR$(N)) GOSUB 30,100,200,300,400,500
.
.
.
```

Also see the VAL function, Section 3.42.

3.38 STRING$

Syntaxes

```
STRING$(I,J)
STRING$(I,X$)
```

Action Returns a string of length I whose characters all have ASCII code J or the first character of X$.

Example

```
10 X$ = STRING$(10,45)
20 PRINT X$ "MONTHLY REPORT" X$
RUN
----------MONTHLY REPORT----------
Ok
```

3.39 TAB

Syntax TAB(I)

Action Spaces to position I on the terminal. If the current print position is already beyond space I, TAB goes to that position on the next line. Space 1 is the left-

most position, and the rightmost position is the width minus one. I must be in the range 1 to 255. TAB may only be used in PRINT and LPRINT statements.

Example

```
10 PRINT "NAME" TAB(25) "AMOUNT" : PRINT
20 READ A$,B$
30 PRINT A$ TAB(25) B$
40 DATA "G. T. JONES","$25.00"
RUN
NAME                      AMOUNT
G. T. JONES               $25.00
Ok
```

3.40 TAN

Syntax TAN(X)

Action Returns the tangent of X in radians. TAN(X) is calculated in single precision. If TAN overflows, the "Overflow" error message is displayed, machine infinity with the appropriate sign is supplied as the result, and execution continues.

Example

```
10 Y=Q*TAN(X)/2
```

3.41 USR

Syntax USR[<digit>](X)

Action Calls the user's assembly language subroutine with the argument X. <digit> is in the range 0 to 9 and corresponds to the digit supplied with the

DEF USR statement for that routine. If <digit> is omitted, USR0 is assumed. See "Assembly Language Subroutines," in the *Microsoft BASIC User's Guide.*

Example

```
40 B = T*SIN(Y)
50 C = USR(B/2)
60 D = USR(B/3)
 .
 .
 .
```

3.42 VAL

Syntax VAL(X$)

Action Returns the numerical value of string X$. The VAL function also strips leading blanks, tabs, and linefeeds from the argument string. For example,

VAL(" -3")

returns -3.

Example

```
10 READ NAME$,CITY$,STATE$,ZIP$
20 IF VAL(ZIP$)<90000 OR VAL(ZIP$)>96699 THEN
PRINT NAME$ TAB(25) "OUT OF STATE"
30 IF VAL(ZIP$)>=90801 AND VAL(ZIP$)<=90815
THEN PRINT NAME$ TAB(25) "LONG BEACH"
 .
 .
 .
```

See the STR$ function, Section 3.37, for details on numeric-to-string conversion.

3.43 VARPTR

Syntax 1 VARPTR(<variable name>)

Syntax 2 VARPTR(#<file number>)

Action *Syntax 1*

Returns the address of the first byte of data identified with <variable name>. A value must be assigned to <variable name> prior to execution of VARPTR. Otherwise an "Illegal function call" error results. Any type variable name may be used (numeric, string, array). For string variables, the address of the first byte of the string descriptor is returned (see "BASIC Assembly Language Subroutines," in the *Microsoft BASIC User's Guide* for discussion of the string descriptor). The address returned will be an integer in the range -32768 to 32767. If a negative address is returned, add it to 65536 to obtain the actual address.

VARPTR is usually used to obtain the address of a variable or array so it may be passed to an assembly language subroutine. A function call of the form VARPTR(A(0)) is usually specified when passing an array, so that the lowest-addressed element of the array is returned.

Note All simple variables should be assigned before calling VARPTR for an array, because the addresses of the arrays change whenever a new simple variable is assigned.

Syntax 2

For sequential files, returns the starting address of the disk I/O buffer assigned to <file number>. For random files, returns the address of the FIELD buffer assigned to <file number>.

Example

```
100 X = USR(VARPTR(Y))
```

Appendices

A Error Codes and Error Messages 131

B Mathematical Functions 137

C ASCII Character Codes 139

D Microsoft BASIC Reserved Words 141

Appendix A

Error Codes and Error Messages

Code	Number	Message
NF	1	NEXT without FOR A variable in a NEXT statement does not correspond to any previously executed, unmatched FOR statement variable.
SN	2	Syntax error A line is encountered that contains some incorrect sequence of characters (such as unmatched parenthesis, misspelled command or statement, incorrect punctuation, etc.). Microsoft BASIC automatically enters edit mode at the line that caused the error.
RG	3	Return without GOSUB A RETURN statement is encountered for which there is no previous, unmatched GOSUB statement.
OD	4	Out of data A READ statement is executed when there are no DATA statements with unread data remaining in the program.
FC	5	Illegal function call A parameter that is out of range is passed to a math or string function. An FC error may also occur as the result of: 1. A negative or unreasonably large subscript.

2. A negative or zero argument with LOG.
3. A negative argument to SQR.
4. A negative mantissa with a noninteger exponent.
5. A call to a USR function for which the starting address has not yet been given.
6. An improper argument to MID$, LEFT$, RIGHT$, INP, OUT, WAIT, PEEK, POKE, TAB, SPC, STRING$, SPACE$, INSTR, or ON...GOTO.

OV 6 Overflow

The result of a calculation is too large to be represented in Microsoft BASIC number format. If underflow occurs, the result is zero and execution continues without an error.

OM 7 Out of memory

A program is too large, or has too many FOR loops or GOSUBs, too many variables, or expressions that are too complicated.

UL 8 Undefined line

A nonexistent line is referenced in a GOTO, GOSUB, IF...THEN...ELSE, or DELETE statement.

BS 9 Subscript out of range

An array element is referenced either with a subscript that is outside the dimensions of the array or with the wrong number of subscripts.

DD 10 Redimensioned array

Two DIM statements are given for the same array; or, a DIM statement is given for an array after the default dimension of 10 has been established for that array.

/0	11	Division by zero A division by zero is encountered in an expression; or, the operation of involution results in zero being raised to a negative power. Machine infinity with the sign of the numerator is supplied as the result of the division, or positive machine infinity is supplied as the result of the involution, and execution continues.
ID	12	Illegal direct A statement that is illegal in direct mode is entered as a direct mode command.
TM	13	Type mismatch A string variable name is assigned a numeric value or vice versa; a function that expects a numeric argument is given a string argument or vice versa.
OS	14	Out of string space String variables have caused BASIC to exceed the amount of free memory remaining. Microsoft BASIC will allocate string space dynamically, until it runs out of memory.
LS	15	String too long An attempt is made to create a string more than 255 characters long.
ST	16	String formula too complex A string expression is too long or too complex. The expression should be broken into smaller expressions.
CN	17	Can't continue An attempt is made to continue a program that: 1. Has halted due to an error.

2. Has been modified during a break in execution.
3. Does not exist.

UF 18 Undefined user function

A USR function is called before the function definition (DEF statement) is given.

19 No RESUME

An error handling routine is entered but contains no RESUME statement.

20 RESUME without error

A RESUME statement is encountered before an error handling routine is entered.

21 Unprintable error

An error message is not available for the error condition which exists.

22 Missing operand

An expression contains an operator with no operand following it.

23 Line buffer overflow

An attempt has been made to input a line that has too many characters.

26 FOR without NEXT

A FOR statement was encountered without a matching NEXT.

29 WHILE without WEND

A WHILE statement does not have a matching WEND.

30 WEND without WHILE

A WEND statement was encountered without a matching WHILE.

Disk Errors

Number	Message
50	Field overflow A FIELD statement is attempting to allocate more bytes than were specified for the record length of a random file.
51	Internal error An internal malfunction has occurred in Microsoft BASIC. Report to Microsoft the conditions under which the message appeared.
52	Bad file number A statement or command references a file with a file number that is not OPEN or is out of the range of file numbers specified at initialization.
53	File not found A LOAD, KILL, or OPEN statement references a file that does not exist on the current disk.
54	Bad file mode An attempt is made to use PUT, GET, or LOF with a sequential file, to LOAD a random file, or to execute an OPEN statement with a file mode other than I, O, or R.
55	File already open A sequential output mode OPEN statement is issued for a file that is already open; or a KILL statement is given for a file that is open.

57 Disk I/O error

An I/O error occurred on a disk I/O operation. It is a fatal error; i.e., the operating system cannot recover from the error.

58 File already exists

The filename specified in a NAME statement is identical to a filename already in use on the disk.

61 Disk full

All disk storage space is in use.

62 Input past end

An INPUT statement is executed after all the data in the file has been INPUT, or for a null (empty) file. To avoid this error, use the EOF function to detect the end-of-file.

63 Bad record number

In a PUT or GET statement, the record number is either greater than the maximum allowed (32,767) or equal to zero.

64 Bad file name

An illegal form is used for the filename with a LOAD, SAVE, KILL, or OPEN statement (e.g., a filename with too many characters).

66 Direct statement in file

A direct statement is encountered while LOADing an ASCII-format file. The LOAD is terminated.

67 Too many files

An attempt is made to create a new file (using SAVE or OPEN) when all 255 directory entries are full.

Appendix B
Mathematical Functions

Derived Functions

Functions that are not intrinsic to Microsoft BASIC may be calculated as follows.

Function	Microsoft BASIC Equivalent
SECANT	SEC(X)=1/COS(X)
COSECANT	CSC(X)=1/SIN(X)
COTANGENT	COT(X)=1/TAN(X)
INVERSE SINE	ARCSIN(X)=ATN(X/SQR(-X*X+1))
INVERSE COSINE	ARCCOS(X)=-ATN(X/SQR(-X*X+1)) +1.5708
INVERSE SECANT	ARCSEC(X)=ATN(X/SQR(X*X-1)) +SGN(SGN(X)-1)*1.5708
INVERSE COSECANT	ARCCSC(X)=ATN(X/SQR(X*X-1)) +(SGN(X)-1)*1.5708
INVERSE COTANGENT	ARCCOT(X)=ATN(X)+1.5708
HYPERBOLIC SINE	SINH(X)=(EXP(X)-EXP(-X))/2
HYPERBOLIC COSINE	COSH(X)=(EXP(X)+EXP(-X))/2
HYPERBOLIC TANGENT	TANH(X)=(EXP(X)-EXP(-X))/ (EXP(X)+EXP(-X))
HYPERBOLIC SECANT	SECH(X)=2/(EXP(X)+EXP(-X))
HYPERBOLIC COSECANT	CSCH(X)=2/(EXP(X)-EXP(-X))
HYPERBOLIC COTANGENT	COTH(X)=(EXP(X)+EXP(-X))/ (EXP(X)-EXP(-X))

INVERSE HYPERBOLIC SINE	ARCSINH(X)=LOG(X+SQR(X*X+1))
INVERSE HYPERBOLIC COSINE	ARCCOSH(X)=LOG(X+SQR(X*X-1))
INVERSE HYPERBOLIC TANGENT	ARCTANH(X)=LOG((1+X)/(1-X))/2
INVERSE HYPERBOLIC SECANT	ARCSECH(X)=LOG((SQR(-X*X+1) +1)/X)
INVERSE HYPERBOLIC COSECANT	ARCCSCH(X)=LOG((SGN(X)*SQR (X*X+1)+1)/X)
INVERSE HYPERBOLIC COTANGENT	ARCCOTH(X)=LOG((X+1)/(X-1))/2

Appendix C
ASCII Character Codes

Dec	Hex	CHR	Dec	Hex	CHR	Dec	Hex	CHR
000	00H	NUL	043	2BH	+	086	56H	V
001	01H	SOH	044	2CH	,	087	57H	W
002	02H	STX	045	2DH	-	088	58H	X
003	03H	ETX	046	2EH	.	089	59H	Y
004	04H	EOT	047	2FH	/	090	5AH	Z
005	05H	ENQ	048	30H	0	091	5BH	[
006	06H	ACK	049	31H	1	092	5CH	\
007	07H	BEL	050	32H	2	093	5DH	]
008	08H	BS	051	33H	3	094	5EH	^
009	09H	HT	052	34H	4	095	5FH	_
010	0AH	LF	053	35H	5	096	60H	'
011	0BH	VT	054	36H	6	097	61H	a
012	0CH	FF	055	37H	7	098	62H	b
013	0DH	CR	056	38H	8	099	63H	c
014	0EH	SO	057	39H	9	100	64H	d
015	0FH	SI	058	3AH	:	101	65H	e
016	10H	DLE	059	3BH	;	102	66H	f
017	11H	DC1	060	3CH	<	103	67H	g
018	12H	DC2	061	3DH	=	104	68H	h
019	13H	DC3	062	3EH	>	105	69H	i
020	14H	DC4	063	3FH	?	106	6AH	j
021	15H	NAK	064	40H	@	107	6BH	k
022	16H	SYN	065	41H	A	108	6CH	l
023	17H	ETB	066	42H	B	109	6DH	m
024	18H	CAN	067	43H	C	110	6EH	n
025	19H	EM	068	44H	D	111	6FH	o
026	1AH	SUB	069	45H	E	112	70H	p
027	1BH	ESCAPE	070	46H	F	113	71H	q
028	1CH	FS	071	47H	G	114	72H	r
029	1DH	GS	072	48H	H	115	73H	s
030	1EH	RS	073	49H	I	116	74H	t
031	1FH	US	074	4AH	J	117	75H	u
032	20H	SPACE	075	4BH	K	118	76H	v
033	21H	!	076	4CH	L	119	77H	w
034	22H	"	077	4DH	M	120	78H	x
035	23H	#	078	4EH	N	121	79H	y
036	24H	$	079	4FH	O	122	7AH	z
037	25H	%	080	50H	P	123	7BH	{
038	26H	&	081	51H	Q	124	7CH	|
039	27H	'	082	52H	R	125	7DH	}
040	28H	(	083	53H	S	126	7EH	~
041	29H	)	084	54H	T	127	7FH	DEL
042	2AH	*	085	55H	U			

Dec=decimal, Hex=hexadecimal (H), CHR=character, LF=Line Feed, FF=Form Feed, CR=Carriage Return, DEL=Rubout

Index

ABS, 107
Addition, 16
ALL, 31, 37
Arctangent, 108
Array variables, 12, 37, 45
Arrays, 12, 35, 39, 51
ASC, 108
ASCII codes, 108, 139
ASCII format, 32, 73, 97
Assembly language subroutines, 30, 44, 79, 128
ATN, 108
AUTO, 6, 29

Boolean operations, 19

CALL, 30
Carriage return, 7, 64, 65, 102-104
Cassette tape, 35, 39
CDBL, 109
CHAIN, 31, 37
Character set, 6
CHR$, 109
CINT, 110
CLEAR, 34
CLOAD, 35
CLOAD*, 35
CLOAD?, 35
CLOSE, 36
Command level, 5
COMMON, 31, 37
Concatenation, 21
Constants, 8
CONT, 39, 68
Control characters, 8
Control-A, 49
COS, 110
CSAVE, 39
CSAVE*, 39
CSNG, 111
CVD, 111
CVI, 111
CVS, 111

DATA, 41
DEF FN, 42
DEFDBL, 12, 43
DEFINT, 12, 43
DEFSNG, 12, 43
DEFSTR, 12, 43
DEF USR, 44
DELETE, 6, 32, 45
DIM, 45
Direct mode, 5, 62, 76
Division, 16
Double precision, 10, 43, 80, 109

EDIT, 6, 46
Edit mode, 8, 46
ELSE, 61
END, 36, 39, 50, 60
EOF, 112
ERASE, 51
ERL, 51
ERR, 51
ERROR, 52
Error codes, 23, 52-53, 131
Error handling, 52, 76
Error messages, 23, 131
Error trapping, 52, 95
Escape, 7, 47
EXP, 112
Exponentiation, 16, 17
Expressions, 16

FIELD, 54
FIX, 113
FOR...NEXT, 56
FRE, 113
Functions, 21, 42, 107, 137

GET, 54, 59
GOSUB, 59
GOTO, 60

HEX$, 114
Hexadecimal, 9, 114

IF...GOTO, 61
IF...THEN, 20, 61
IF...THEN...ELSE, 61
Indirect mode, 5
INKEY$, 114
INP, 115
INPUT, 53, 64, 67
INPUT#, 65
INPUT$, 115
INSTR, 116
INT, 116
Integer, 110, 113, 116
Integer division, 16

KILL, 66

LEFT$, 117
LEN, 117
LET, 66
Line feed, 6, 64, 67, 103-104
LINE INPUT, 67
LINE INPUT#, 68
Line numbers, 5-6, 30, 93
Line printer, 70, 71, 102, 118
Lines, 5
LIST, 6, 69
LLIST, 70
LOAD, 71, 97
LOC, 117
LOG, 118
Logical operators, 19
Loops, 57, 101
LPOS, 102, 118
LPRINT, 71, 102
LPRINT USING, 71
LSET, 72

MERGE, 31, 73
MID$, 73, 119
MKD$, 119
MKI$, 119
MKS$, 119
MOD operator, 17
Modulus arithmetic, 17
Multiplication, 16

NAME, 74
Negation, 16
NEW, 75
NULL, 75
Numeric constants, 8
Numeric variables, 11

OCT$, 120
Octal, 10, 120
ON ERROR GOTO, 76
ON...GOSUB, 76
ON...GOTO, 76
OPEN, 36, 54, 77
Operators, 15, 18-22
OPTION BASE, 78
OUT, 79
Overflow, 17, 112, 126
Overlay, 32

Paper tape, 75
PEEK, 79, 120
POKE, 79, 121
POS, 102, 121
PRINT, 80
PRINT USING, 82
PRINT#, 86
PRINT# USING, 86
Protected files, 97
PUT, 54, 89

Random files, 54, 59, 66, 72, 77, 89, 118
Random numbers, 90, 122
RANDOMIZE, 90, 122
READ, 91, 94
Relational operators, 18
REM, 31, 92
RENUM, 32, 52, 93
Reserved words, 141
RESTORE, 94
RESUME, 95
RETURN, 60
RIGHT$, 121
RND, 90, 122
RSET, 72
Rubout, 7, 21, 47
RUN, 96

SAVE, 71, 97
Sequential files, 65, 66, 68, 77, 86, 104, 118
SGN, 122
SIN, 123
Single precision, 10, 43, 80, 110
Space requirements for variables, 13
SPACE$, 123
SPC, 124
SQR, 124
STOP, 39, 50, 60, 98
STR$, 125
String constants, 8
String functions, 111, 114-117, 119, 120, 121, 123, 125, 127
String operators, 21
String space, 34
String variables, 11, 43, 67-68
STRING$, 125
Subroutines, 30, 59
Subscripts, 12, 45, 78
Subtraction, 16
SWAP, 99

TAB, 125
Tab, 7, 8
TAN, 126
TROFF, 99
TRON, 99

USR, 44, 126

VAL, 127
Variables, 11
VARPTR, 128

WAIT, 100
WEND, 101
WHILE, 101
WIDTH, 102
WIDTH LPRINT, 102
WRITE, 103
WRITE#, 104